FORMAL SOFTWARE DEVELOPMENT USING Z AND THE REFINEMENT CALCULUS

DENNIS JU-XIENG WEE

FORMAL SOFTWARE DEVELOPMENT USING Z AND THE REFINEMENT CALCULUS

BY

© Dennis Ju-Xieng Wee

A thesis submitted to the School of Graduate

Studies in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

July 1993

St. John's Newfoundland

Abstract

This thesis is a study of a formal software development process that uses a formal specification language called Z [42] and the formal development method called the *refinement calculus* [31]. The software development process is divided into five stages: formal specification in Z, data refinement, translation into the refinement calculus, operation refinement, and translation into the target programming language [25]. In this thesis, many of the important results for understanding and using this process are collected together and numerous examples are given to illustrate their use. Through a case study of the *Paragraph Problem* [5, 31], we show how formality may be appropriately employed to manage the algorithmic complexity in a development, and indicate directions on how predefined programming language and library routines may be introduced into a formal development. The thesis concludes with some suggestions for further research.

Acknowledgments

I would like to thank Helmut Roth and He Xu for their companionship during these two years of sharing an office. It must have been hard for them to put up with my quirks and idiosyncrasies.

I would also like to thank Robert Machin for proofreading a draft of this thesis, and John Rochester and Patrick Martin for helping with some of the typesetting.

I would like to acknowledge Dr. Tony Middleton, the Department of Computer Science, and the School of Graduate Studies for providing the financial support during the course of my M.Sc. degree.

Most sincere thanks to Todd Wareham for his many useful suggestions, criticisms, and comments. His willingness to read this thesis at a very short notice is greatly appreciated.

I am also grateful to the technical and administrative staff at the Department of Computer Science for creating and maintaining a conducive environment for learning. In particular, the Chair of Computer Science, Dr. Paul Gillard, has helped and encouraged me in more ways than I can remember.

I am most grateful to my thesis supervisor, Dr. Tony Middleton, for introducing me to the world of formal specification. I cherish his patience, understanding, sound advice, and humor. His greatness is apparent from his ability to put up with my obstinacy.

Last but not least, I must thank my favorite musicians: Alfred Brendel, Zino

Francescatti, Murizio Pollini, Pascal Rogé, Andras Schiff, Rudolf Serkin. and Tamás Vásáry, for contributing to my mental stability. But perhaps most of all, I am indebted to the late Claudio Arrau for his pianistic legacy; if it were not for his Beethoven, Chopin, Debussy, Mozart, Lizst, and Schumann, I would not have been able to retain my sanity during the last month of thesis writing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal methods in software development are mathematical techniques which may be used to specify, develop and verify software systems in a systematic and organized fashion. The mathematical basis of a formal method is, in principle, given by a *formal specification language* with a well-defined syntax and semantics.

1.1 Formal Methods in Software Development

Some of the advantages of using formal methods in software development are given below.

1.1.1 Formal Specification

A formal method is commonly used to specify software systems. Its basis language is used as a notation to write formal specifications. Since the notation is precise, the resulting formal description is clear and unambiguous.

There are several advantages to using formal rather than informal languages to specify software. With an informal specification, thorough reasoning is often hard or impossible; a formal specification, on the other hand, may be subjected to rigorous mathematical analysis which easily exposes *ambiguities* and *incompleteness*. Since a formal specification is essentially a mathematical theory, its *consistency* can also be checked. An inconsistent specification is undesirable since it contains contradicting facts [44] and a program based on it cannot be realized. The mathematical nature of a formal specification also lets the specifier formally prove important properties of the system to the customer, thereby ensuring that the specifier has a good approximation of the customer's requirements for the system.

1.1.2 Formal Development

A program may be mathematically derived from the program's formal specification. A program derived in this manner is guaranteed to satisfy its description.

One such development method called *refinement* involves developing programs in small steps. A step may consist of defining a module as a collection of modules at a lower level, or choosing a representation for a data type that is more efficient or more easily constructable in the target programming language. Starting from a specification, each refinement step yields another specification that contains more implementation details. This latter specification must in turn be shown to satisfy the former in order to ensure correctness. Such proof of satisfaction often generates proof obligations which can be precisely stated and discharged within the framework of a formal method [44].

1.1.3 Verification versus Validation

Following Wing [44] and Hayes and Jones [17], a *verification* is a formal proof that an implementation satisfies its specification, while a *validation* is an informal check of correctness, e.g., testing. When a program is not formally developed, it may be desirable to verify its correctness. Only when the specification is expressed mathematically can a formal proof be carried out; without such a specification, only validation is possible [44, 17].

An in-depth discussion of the merits of formal methods is not an objective of this thesis; the interested reader is referred to [15, 26, 44]. From here onwards, we concern ourselves with a software development process that relies on formal methods [25].

1.2 A Formal Development Process

A software development process that uses the formal specification language Z, and the formal development method called the *refinement calculus*, is described in [25, 45]. This process (see Figure 1.1) may be viewed as having five stages:

$$\begin{tikzcd}
{} \arrow[d, "\text{Formal specification in Z}"] \\
\text{Abstract Specification} \arrow[d, "\text{Data refinement}"] \\
\text{Concrete Design} \arrow[d, "\text{Translation into the refinement calculus}"] \\
\text{Abstract Program} \arrow[d, "\text{Operation refinement}"] \\
\text{Code (Guarded Commands)} \arrow[d, "\text{Translation into the target programming language}"] \\
\text{Code (Pascal, C, ...)}
\end{tikzcd}$$

Figure 1.1: Stages of software development using Z and the *refinement calculus*.

formal specification in Z, data refinement, translation into the refinement calculus, operation refinement, and translation into the target programming language. An overview of these stages is given next.

1.2.1 Formal Specification in Z

The Z notation [42] is used to formally specify the proposed system. The formal specification obtained is called an *abstract specification* as it contains abstract mathematical models of data types and operations. Although these models are typically difficult to construct using the primitive data types of the target programming language, they are well suited for describing and reasoning about the properties of the system.

In Chapter 2, a brief account of the Z specification language and a convention for specifying software systems is given. This exposition is illustrated by a case study in which some operations of the abstract data type *stack* are specified.

1.2.2 Data Refinement

Data refinement is the process of transforming an abstract specification into a specification of the system which contains data types that are either available or easily constructed in the target programming language. The product of this refinement is called a *concrete design* since it uses data types that may be directly realized in the target programming language. An important task here is to formulate a *retrieve relation* to relate the abstract specification and the concrete design. Proof obligations which use this relation may be discharged to show that this concrete design satisfies the abstract specification.

The process of producing a concrete design from an abstract specification is the subject of Chapter 3. The purpose of data refinement is illustrated through several examples and the case study of the stack started in Chapter 2.

1.2.3 Translation into the Refinement Calculus

The concrete design is then translated into the notation of the *refinement calculus* [31] to obtain an *abstract program*. While the Z notation is more suitable for the purpose of specification, the refinement calculus is more appropriate for program development.

The necessity of and strategies for translation are discussed in Chapter 4. Rules are formulated to allow the translation process to be performed in a straightforward manner. These rules indicate how the common structures in a Z specification may be transformed into the refinement calculus.

1.2.4 Operation Refinement

Code written in a language based on Dijkstra's *guarded commands* [13] is calculated from the design by performing refinement steps. These steps are carried out according to the laws of the refinement calculus, which guarantee that the derived code satisfies its specification.

Some elementary laws of the refinement calculus are given in Chapter 5. Examples including the stack case study are presented to illustrate their use.

1.2.5 Translation into the Target Programming Language

Since the stages of data and operation refinement take into consideration the characteristics of the target programming language, the resulting code is reasonably close to allow a simple and intuitive conversion into the target programming language. Hence, the code from the previous step may be easily translated into an imperative programming language like C or Pascal.

Due to its language specificity and relative ease, a review of this stage is not given. However, in Chapter 6, the translation of some guarded commands into Pascal may be observed.

1.3 An Application

In Chapter 6, the formal software development process described here is used to produce a program for computing *even paragraphs* [5, 31]. An aim of constructing this program is to collect useful experience that may be employed to construct larger and more complicated programs. Besides illustrating many of the concepts that are contained in the earlier part of this thesis, this case study also shows how formality may be appropriately exploited to manage the complexity of the refinement which may arise during the development of a software system. Since this program uses predefined routines, we also give directions on how these may be integrated into the formal development framework.

1.4 Summary

This thesis reports on the practical aspects of a software development process that uses Z and the refinement calculus. The aim is to collect together in one place many of the important theoretical results that are needed to understand a ıd use such a development process. Each stage of the process is documented in a chapter with examples to illustrate its purpose. This thesis concludes with a non-trivial case study and suggestions for future research.

Chapter 2

Formal Specification in Z

Z is a formal specification language based on typed set theory and first-order predicate calculus [19, 40, 42]. This chapter presents some of the features of Z, and how Z may be used to specify software systems in the standard convention as described in [42]. Since a complete description of the notation is not possible, a glossary is included in Appendix A.

2.1 Schemas

Central to Z is a language construct called a *schema* which may be diagrammatically represented in two equivalent ways: vertically and horizontally. A schema named *Schema* written vertically is as follows.

$$\begin{array}{|l}
\hline
\mathit{Schema} \\
v_1 : T_1 \\
v_2 : T_2 \\
\vdots \\
v_n : T_n \\
\hline
P_1 \\
P_2 \\
\vdots \\
P_k \\
\hline
\end{array}$$

A schema consists of two parts: the *declaration* and the *predicate*. The declaration is contained in the part of a schema above the dividing line, which, in the case of *Schema*, has *variables* $v_1, v_2, ..., v_n$, of *types* $T_1, T_2, ..., T_n$. These variables are also known as the *components* of the schema.

Below the line are *predicates* $P_1, P_2, ..., P_k$, which are implicitly conjuncted ("anded") to give the relation which must hold among the values of the variables. The predicate part of a schema may be empty, in which case, it is a box with no dividing line, containing only the signature.

The same schema is written horizontally as follows.

$$\mathit{Schema} \mathrel{\widehat{=}} [v_1 : T_1;\ v_2 : T_2;\ ...;\ v_n : T_n \mid P_1 \wedge P_2 \wedge ... \wedge P_k]$$

2.2 States

The style of Z specification used here is suitable for sequential, imperative programming and it involves viewing a software system as an *abstract data type*. Simply put, an abstract data type consists of a set of states, called the *state* *space*, a non-empty set of *initial states*, and a number of *operations* which transform one state into another [42]. In this section, we show how the state space of a system may be defined.

2.2.1 Sets, Types and Basic Types

The specification of a state space involves identifying some objects of interest. Each such object has a type which is composed from sets. Z contains standard mathematical sets like the natural numbers $\mathbb{N}$ and the integers $\mathbb{Z}$, etc. In general, any set may be used as a type, and complex types like sequences and cartesian products may be constructed from simpler ones by using standard Z operators.

A particularly useful construction in Z is that of a *basic type* which allows a set to be declared without mentioning what is contained in it. The declaration

$[OBJECT]$

indicates the existence of a set of objects called $OBJECT$, although we do not know its structure or content.

2.2.2 Axiomatic Descriptions

Global constants and functions may be declared and defined using *axiomatic descriptions*. These descriptions allows the declaration and use of global variables. The scope of a global variable extends from the point of declaration to the end of the specification.

$$\begin{array}{|l} max : \mathbb{N} \\ \hline max = 20 \end{array}$$

For example, a global variable *max* of type natural number is declared. A constraint on its value is included, which restricts *max* to a value of 20.

2.2.3 Modeling States

The *state space* of a system is the set of allowable states. This set may be defined with a schema by declaring *state variables* as components of the schema and constraining their values using the schema predicates. The conjunction of these predicates gives the system *invariant*, and the values that may be taken up by the variables represent the allowable states of the system. For example, a possible state space of a system that maintains a rather limited version of the abstract data type *stack* is

$$\begin{array}{|l} \textit{Stack} \\ \hline stack : \mathrm{seq}\, OBJECT \\ \hline \# stack \leq max \\ \hline \end{array}$$

The schema *Stack* models a stack which may be used to store objects from the set *OBJECT*. It has a state variable *stack* which is a finite sequence (seq) of *OBJECT*, and its invariant requires that the length of the stack be not more than 20. In this paper, the convention of writing schema names with the first letter capitalized, and component names with the first letter in lower case is used.

2.3 Initial States

The *initial states* of a system may be documented by describing the values that the state variables must take when the system is started up. A system typically has only one such state, but there may be more. The initial state of our stack system is given in *InitStack*.

```
┌─ InitStack ──────────────────────────────
│ stack′ : seq OBJECT
├──────────
│ #stack′ ≤ max
│ stack′ = ⟨⟩
└──────────────────────────────────────────
```

The significance of the dash (′) is explained in a later section. Since ⟨⟩ is the empty sequence, *InitStack* requires that the stack is initially empty.

2.3.1 Schema Reference

The *InitStack* schema may be rewritten using a mechanism called *schema reference* which enables Z specifications to be structured in a modular fashion. Below, two features of this mechanism, *decoration* and *inclusion*, are described.

<u>Systematic Decoration</u>

Within the revised version of *InitStack* shown below, the schema name *Stack* appears with a prime ('); this is an operation on schemas called *decoration*. Essentially, any decoration that is applied on the name of a schema is inherited by its components.

<u>Schema Inclusion</u>

By including *Stack'* in *InitStack*, the variables and predicates of the former are included in the declaration and predicate parts of the latter; the variables are merged and the predicates are conjuncted.

Using these features, the schema *InitStack* may be alternatively and more economically specified as

```
─ InitStack ──────────────────────────────────
│ Stack'
├──────────────
│ stack' = ⟨⟩
└─────────────────────────────────────────────
```

2.3.2 Showing Existence of Initial States

It is meaningful to check that an initial state does exist, and we may do so by first expressing it as a theorem.

$$\exists\, Stack' \bullet InitStack$$

This is equivalent to proving

$$\exists\, stack' : \mathrm{seq}\, OBJECT \bullet \#stack' \leq max \land stack' = \langle\rangle$$

which is trivially true when *stack'* is an empty sequence.

2.4 Operations

An operation is modeled as a *state change* by declaring a schema containing *before-* and *after-state variables*, which indicate the states of the system before and after the operation has taken place. By convention, the before-variables are unprimed while the after-variables are primed ($'$), and the state change of an operation is specified by describing the relationship between these variables.

2.4.1 The Δ and Ξ Conventions

Before specifying any operation, it is convenient to write schemas that suggest a possible change and no change in the state of the system. By convention, the names of these schemas start with Δ and Ξ respectively.

```
─ ΔStack ──────────────────────────────
│ Stack
│ Stack'
└──────────────────────────────────────
```

The schema $\Delta Stack$ suggests a change of the stack since the schema does not contain any predicate to constrain the values of the state variables.

```
─ ΞStack ──────────────────────────────
│ Stack
│ Stack'
├──────────────
│ stack' = stack
└──────────────────────────────────────
```

The schema $\Xi Stack$ indicates a no change during the operation since the schema contains a predicate that requires the after-value of the stack be the same as its before-value. These schemas are useful as short-hands for specifying operations on the stack.

2.4.2 Specifying Operations

Using $\Delta Stack$ and $\Xi Stack$, the *push*, *pop*, and *top* operations of the stack may now be succinctly specified.

Pushing an Element onto the Stack

The symbol $\frown$ is the operator for sequence concatenation, and $\langle object? \rangle$ is the sequence containing only *object?*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PushOk}} \\ \hline
\Delta Stack \\
object? : OBJECT \\ \hline
\# stack < max \\
stack' = stack \frown \langle object? \rangle \\ \hline
\end{array}$$

The schema *PushOk* describes the operation of pushing *object?* onto a stack. The variables in *PushOk* consist of the before- and after-variables which are included with $\Delta Stack$, and an input variable *object?* which, by convention, ends with a question mark.

It is often recommended that the specification of an operation document explicitly the *precondition*, which states the condition under which the operation may be used. Typically, the precondition appears as the first predicate in the schema. For *PushOk*, this requires that the stack contains less than *max* elements, i.e., the stack must not be full.

The actual push operation is described as the after-stack being the same as the before-stack with the input *object*? concatenated to its end.

Popping an Element off the Stack

PopOk

$\Delta Stack$

$stack \neq \langle\rangle$

$stack' = front\ stack$

The Z specification language includes a *mathematical toolkit* which is a collection of predefined mathematical types and primitives that allows specifications to be built in a compact way. For sequences, the toolkit contains a function *front* that takes a non-empty sequence and returns the same sequence with the last element removed. Using *front*, popping an element off the stack is described as taking away its last element.

Inquiring the Top Element of the Stack

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TopOk}} \\
\hline
\Xi \mathit{Stack} \\
\mathit{object}! : \mathit{OBJECT} \\
\hline
\mathit{stack} \neq \langle\rangle \\
\mathit{object}! = \mathit{last}\ \mathit{stack} \\
\hline
\end{array}$$

The schema *TopOk* describes the operation of reporting the value of the top element in a non-empty stack. The requirement that the stack not be changed is stated by including Ξ*Stack*. The operation is specified using the *last* operator, which takes a non-empty sequence and returns the value of the last element of the sequence. This value is recorded in the output variable *object*! which, by convention, ends with an exclamation mark.

2.5 Preconditions

The precondition of an operation must be properly documented since it states exactly when an operation should be used. When an operation is invoked under its precondition, the specification requires that it terminates in a state that satisfies the predicates written in the schema; otherwise, it does not say what is to happen, i.e., the operation's result is unpredictable.

The precondition of an operation describes all those before-states from which an after-state is guaranteed. Often, an implementation of an operation assumes that its precondition holds on the before-states, which means that the resulting program may be used appropriately only under the circumstances depicted in the precondition. This stresses the importance of correctly documenting the precondition [46].

2.5.1 Calculating Preconditions

In Z, the precondition of an operation Op is denoted pre Op, and is calculated by *hiding* the after-state and output variables. This is accomplished by existentially quantifying these variables in the predicate part of Op. As an illustration, the precondition of the operation Op is calculated below.

```
┌─ State ──────────────────────
│ v : V
├──────────
│ inv
└──────────────────────────────
```

```
┌─ Op ─────────────────────────
│ ΔState
│ x? : X
│ y! : Y
├──────────
│ Pred
└──────────────────────────────
```

Assuming that $State$ is the state schema of the system, pre Op is the schema obtained by existentially quantifying the after- and output variables v' and $y!$.

```
┌──────────────────────────────
│ State
│ x? : X
├──────────
│ ∃ State'; y! : Y • Pred
└──────────────────────────────
```

When mentioning the precondition of an operation, we commonly refer to the predicate in the precondition schema of the operation. In the case of Op, this is

$$\exists State'; \; y! : Y \bullet Pred$$

which is equivalent to

$$\exists v' : V; \; y! : Y \mid inv' \bullet Pred$$

where inv' is the state invariant with all the state variables primed[1].

2.5.2 Simplifying Preconditions

Preconditions calculated in this way often contain extraneous details which may be easily eliminated. Woodcock suggests two strategies for simplifying these predicates [46].

The One-Point Rule

The first tactic uses the so-called *one-point rule* which states that the definition of a variable may be substituted for the variable itself. In symbols, this may be expressed as

$$(\exists x : S \bullet P(x) \land x = term) \quad \Leftrightarrow \quad P(term)$$

with the condition that x is not free in $term$.

[1] Note that the use of the dash ($'$) for inv is not standard.

For simplifying preconditions, this rule is often used when an output or after-variable has an equality constraining its value. This value may be systematically substituted for all its occurrences and its quantification is then dropped.

The Conditional-Rewrite Rule

The second tactic is summarized in the following *conditional-rewrite rule*.

$$\frac{P \Rightarrow Q}{(P \land Q) \quad \Leftrightarrow \quad P}$$

This rule says that, for predicates P and Q, if $P \Rightarrow Q$ is true, then $P \land Q$ may be rewritten as P.

Simplifying the Precondition of *PopOk*

The precondition of *PopOk* is calculated and simplified using the one-point and conditional-rewrite rules as shown below. By definition, pre *PopOk* is

$$\begin{aligned}&\exists\, stack' : \mathrm{seq}\, OBJECT \bullet \\ &\qquad \#stack' \leq max \land stack \neq \langle\rangle \land stack' = front\ stack.\end{aligned}$$

Since *stack* is free, it may be moved outside the quantification, and we have

$$\begin{aligned}\Leftrightarrow\ &(\exists\, stack' : \mathrm{seq}\, OBJECT \bullet \\ &\qquad stack' = front\ stack \land \#stack' \leq max) \land stack \neq \langle\rangle.\end{aligned}$$

Using the one-point rule, *stack'* may be substituted with its definition of *front stack*, and we have

Operation	Precondition
$PushOk$	$\#stack < max$
$PopOk$	$stack \neq \langle\rangle$
$TopOk$	$stack \neq \langle\rangle$

Table 2.1: The preconditions of $PushOk$, $PopOk$, and $TopOk$.

$$\Leftrightarrow \quad \#(front\ stack) \leq max \land stack \neq \langle\rangle.$$

From the system invariant, we know that $\#stack \leq max$; therefore, it is easily proved that $stack \neq \langle\rangle \Rightarrow \#(front\ stack) \leq max$. Using this in conjunction with the conditional-rewrite rule, the predicate $\#(front\ stack) \leq max \land stack \neq \langle\rangle$ may be simplified as $stack \neq \langle\rangle$, and the final step of our proof is

$$\Leftrightarrow \quad stack \neq \langle\rangle.$$

Similarly, the preconditions for $PushOk$ and $TopOk$ are calculated and they are collected in Table 2.1.

2.6 Proving Properties of Systems

As mentioned in the previous chapter, a formal specification may be used to prove important properties of the system. In this section, we describe how the last-in-first-out property of the stack may be shown. This uses the sequential composition operator $\fatsemi$ which is described next.

Sequential Composition

The sequential composition of two operation schemas, Op_1 and Op_2, may be understood as a schema describing the operation of performing first Op_1 and then Op_2. The schema $Op_1 \fatsemi Op_2$ is obtained by "combining" Op_1 and Op_2, where the after-variables of Op_1 and the before-variables of Op_2 are both equated with some intermediate state variables. If $State$ is the schema describing the system state, $Op_1 \fatsemi Op_2$ is defined as

$$
\begin{array}{l}
\exists\, State'' \bullet \\
\quad (\exists\, State' \bullet [Op_1;\ State'' \mid \theta State' = \theta State'']) \land \\
\qquad (\exists\, State \bullet [Op_2;\ State'' \mid \theta State = \theta State''])
\end{array}
$$

where $\theta State$ may be thought of as the tuple formed from the state variables [42].

Showing the Last-In-First-Out Property of the Stack

The last-in-first-out property of the stack may be shown by proving that the stack is restored to its original content in a sequence of $PushOk$ and $PopOk$ operations, provided that the stack is not full to begin with. In symbols, this is

$$
\begin{array}{l}
\forall\, Stack, Stack' \mid \#stack < max \bullet \\
\quad PushOk \fatsemi PopOk \Rightarrow stack = stack'.
\end{array}
$$

Assuming the invariants in $Stack$ and $Stack'$, and the condition $\#stack < max$, the proof may proceed with stating

$$PushOk \fatsemi PopOk$$

which, by definition, is equivalent to

$$\begin{aligned}\Leftrightarrow\ & \exists\, Stack'' \bullet \\ & \quad (\exists\, Stack' \bullet [PopOk;\ Stack'' \mid stack' = stack'']) \land \\ & \quad (\exists\, Stack \bullet [PushOk;\ Stack'' \mid stack = stack'']).\end{aligned}$$

After multiple applications of the one-point and conditional-rewrite rule, we arrive at

$$\Leftrightarrow\ stack \neq \langle\rangle \land stack' = stack$$

which may be simplified as

$$\Leftrightarrow\ stack' = stack$$

since, by hypothesis, $stack \neq \langle\rangle$ is true.

2.7 Errors

The schemas *PushOk*, *PopOk*, and *TopOk* describe only successful operations. For instance, for *PushOk*, the specification says what happens when the stack is not full, but it does not indicate what the program should do if it is full. In this sense, the operations are *incomplete*.

Sometimes, it is desirable and possible to specify operations so that they are more applicable, and this often requires the specification to include what should happen when an operation is invoked under conditions for which it is not intended. Typically, this is achieved by making the operation do some sort of *error handling*.

2.7.1 Reporting Errors

The operations of the stack can be modified so that the status of the execution of each operation is reported in a variable *result*!. Three types of messages are used: *ok* to signify a successful operation, *empty* and *full* to report empty and full stack respectively.

Free Type Definitions

A *free type definition* allows Z to define a set with certain objects. This is very useful for defining a type and its elements. For example, we may define the set *REPORT* consisting of three elements *ok*, *empty*, and *full* with the following free type definition.

$$REPORT ::= ok \mid empty \mid full.$$

Reporting a Successful Operation

The set *REPORT* may now be used in the schema *Success*, which describes the operation of reporting a successful operation.

```
_Success_________________________________
| result! : REPORT
|--------------
| result! = ok
|________________________________________
```

Reporting a Full Stack

For example, we can report a full stack as follows.

$$\begin{array}{|l}
\hline \mathit{StackFull} \\
\Xi \mathit{Stack} \\
\mathit{result}! : \mathit{REPORT} \\
\hline \#\mathit{stack} = \mathit{max} \\
\mathit{result}! = \mathit{full} \\
\hline
\end{array}$$

In *StackFull*, *result!* is given the value *full* when the stack reaches its maximum capacity. It further requires that there should be no change in the stack.

<u>**Reporting an Empty Stack**</u>

Similarly, reporting an empty stack can be written as

$$\begin{array}{|l}
\hline \mathit{StackEmpty} \\
\Xi \mathit{Stack} \\
\mathit{result}! : \mathit{REPORT} \\
\hline \mathit{stack} = \langle\rangle \\
\mathit{result}! = \mathit{empty} \\
\hline
\end{array}$$

2.7.2 Schema Calculus

One of the powerful features of Z that makes it appropriate for writing specifications of large systems is its *schema calculus* which enables larger schemas to be formed by combining smaller schemas using *schema connectives*. In the following, two of these connectives, $\wedge$ and $\vee$, are used to build a stronger specification of the stack operations. Using the $\wedge$ operator on two schemas merges their declarations and conjuncts their predicates, while the $\vee$ operation has the same effect except that the predicates are disjuncted.

Schema connectives are useful operators in that they allow parts of a specification to be considered separately. For instance, for our stack, the specifications of successful operations and error handling are considered separately and these are then combined, using schema connectives, to form a more complete specification.

2.7.3 Building Stronger Specifications

Using schema definition ($\widehat{=}$), the new schema *Pop* is formed, first by making a *schema expression* from the conjuncting of *PopOk* and *Success*, which is then disjuncted with *StackEmpty*.

$$Pop \mathrel{\widehat{=}} (PopOk \wedge Success) \vee StackEmpty$$

The schema *Pop* is made explicit below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Pop}} \\
\hline
Stack \\
Stack' \\
result! : REPORT \\
\hline
((stack \neq \langle\rangle \wedge \\
\qquad stack' = front\ stack \wedge \\
\qquad result! = ok) \\
\vee \\
(stack = \langle\rangle \wedge \\
\qquad stack' = stack \wedge \\
\qquad result! = empty)) \\
\hline
\end{array}
$$

The specification says that when the stack is not empty, it is popped and a message indicating a successful operation is reported, and that when the stack is empty, it stays the same during the operation and a message indicating an empty stack is reported. Similar schemas for the push and pop operations are defined as

$$\begin{aligned} Push &\mathrel{\widehat{=}} (PushOk \land Success) \lor StackFull \\ Top &\mathrel{\widehat{=}} (TopOk \land Success) \lor StackEmpty \end{aligned}$$

Preconditions Revisited

It would be convenient if the preconditio.ı of the larger schemas could be calculated from the preconditions of the smaller ones from which it is built. In this section, we give a few sugges'.ions on how this may be done.

Since the existential quantification distributes through disjunction, the precondition operator distributes through disjunction as well. Hence, the following equivalence is true.

$$\text{pre}\,(Op_1 \lor Op_2) \Leftrightarrow \text{pre}\ Op_1 \lor \text{pre}\ Op_2$$

The situation is not so simple in the case of conjunction since the existential quantification does not generally distribute through conjunction. However, if the predicates in Op_1 and Op_2 are P_1 and P_2, and the variables contained in P_1 are disjoint from those in P_2, a similar equivalence may be established.

$$\text{pre}\,(Op_1 \land Op_2) \Leftrightarrow \text{pre}\ Op_1 \land \text{pre}\ Op_2$$

Operation	Precondition
StackFull	$\#stack = max$
StackEmpty	$stack = \langle\rangle$
Pop	*true*
Push	*true*
Top	*true*

Table 2.2: The preconditions of *StackFull*, *StackEmpty*, *Pop*, *Push*, and *Top*.

Using these results, the preconditions for the remaining operations are calculated and recorded in Table 2.2. Note that the preconditions of *Pop*, *Push*, and *Top* are all *true*, implying that they may be invoked in any state in the state space of the system; such operations are known as *total* operations.

2.8 Summary and Bibliographical Notes

In this chapter, we have attempted to give a practical guide to the Z specification language. In particular, we have presented a convention of specification which views a system as an abstract data type. Useful information on proving system properties, calculating preconditions, and error-handling is also given.

2.8.1 Some Uses of Z

In recent years, there have been numerous reports of the successful use of Z [8, 43]. In the following, we highlight some of these recent efforts.

Specifying New Systems

Z has been used to describe the development of both software and hardware systems [3, 11, 12]. In [6], Z is used not only to design network services, it is also used to produce the documentation. Bowen indicated that the use of formal methods can lead to a simpler design and more thorough documentation [6].

Specifying Existing Systems

By the specification of existing systems, Z has also been useful in revealing inconsistency and incompleteness. In the post-hoc specification of a real-time kernel, Spivey discovered a design error which could have been easily avoided by using formal techniques [41]. The specification of window systems by Bowen revealed omissions and ambiguities in the documentation [7, 9].

Prototyping

The existence of a formal syntax and semantics for Z implies that it may be amenable to machine analysis and manipulation. This suggests that Z, or a subset of it, in conjunction with an *animator* could be used as a *prototyping* tool. Although there are some arguments against making specifications executable [17], there has been some effort to provide Z with an animator [14, 23].

Testing

Even when a program is mathematically calculated from a formal specification, unless the development steps are guaranteed to have been performed correctly, there is always a need to perform *testing*. Hayes and Hall suggest some techniques for testing based on Z specifications [18, 16]. Hall also discusses the possibility of automatically generating test cases from specifications written in Z [16].

Chapter 3

Data Refinement

The specification in Chapter 2 models a stack with a sequence. Although mathematical data types, like sequences, are very expressive, their operators may not be readily available in the target programming language. This chapter shows how, using data refinement, data types that are more suitable for implementation may be introduced into the specification of a system.

3.1 From Specifications to Designs

In our approach to software development, the task of producing a concrete design from an abstract specification is known as *data refinement*. A procedure for data refining an abstract specification in Z is given in [42, 45]. This involves proposing concrete states and operations, and proving that they satisfy the abstract specification.

3.1.1 Abstract Specifications

Specifications like the one in the previous chapter are *abstract specifications* since they contain data types which usually are not directly implementable. Together with their predefined operators, these data types allow the features of software systems to be described compactly. Furthermore, since their mathematical properties are well-known, they allow easy comprehension of and reasoning about the characteristics of systems.

Although abstract specifications are useful in providing a good understanding of the system, they are generally not good sources from which to produce an implementation directly. This is so because they contain mathematical data types which are inefficient, or are not easily constructable in the target programming language.

Example 3.1 Consider a system that is used to calculate the maximum of a set of integers, whose state space and initial states may be specified as *Max* and *InitMax*.

Max
$numbers : \mathbb{P}\,\mathbb{Z}$

InitMax
Max'
$numbers' = \{\}$

The set of integers maintained by the system is contained in *numbers* where $\mathbb{P}$

is the power set operator, and $\mathbb{P}\,\mathbb{Z}$ is the set of all sets of integers. Operations for entering a number and finding the maximum are described in *Enter* and *FindMax* respectively.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Enter}} \\
\hline
\Delta \mathit{Max} \\
\mathit{number?} : \mathbb{Z} \\
\hline
\mathit{numbers}' = \mathit{numbers} \cup \{\mathit{number?}\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindMax}} \\
\hline
\Xi \mathit{Max} \\
\mathit{maximum!} : \mathbb{Z} \\
\hline
\mathit{numbers} \neq \{\} \\
\mathit{maximum!} = \mathit{max}\ \mathit{numbers} \\
\hline
\end{array}
$$

□

The operations in Example 3.1 are described using the set operators $\cup$ (set union) and *max* (maximum number in a set). Since the properties of sets and their operators are familiar to many, the features of the system may be understood quickly and clearly.

Although sets are very expressive, they are not readily available in some programming languages (e.g., C). The system as specified above also has an inefficiency: since we are only interested in the maximum of the set, there is no need to store the other numbers. To overcome this inefficiency, another specification called a design may be produced.

3.1.2 Concrete Designs

Like an abstract specification, a *concrete design* gives a description of the system; however, it also contains data types that are oriented towards computer processing. The states and operations described in a design are concrete since they can be realized in the target programming language.

In the next example, we show how the concrete states and operations of a concrete design may be proposed.

Example 3.2 Assuming that the target programming language allows boolean and integer variables to be declared, a concrete design for the abstract specification of Example 3.1 is given below. The concrete state space and initial states of the system are described in *MaxC* and *InitMaxC*, respectively.

$BOOLEAN ::= true \mid false$

$$\begin{array}{|l}
\text{\_\_ } MaxC \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
maxNumber : \mathbb{Z} \\
setEmpty : BOOLEAN \\
\hline
\end{array}$$

$$\begin{array}{|l}
\text{\_\_ } InitMaxC \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
MaxC' \\
\hline
setEmpty' = true \\
\hline
\end{array}$$

As mentioned previously, the system needs to keep track of only one number, which the concrete version stores in the integer variable *maxNumber*. The system also maintains a boolean variable *setEmpty* to indicate whether any number has been input into the system. Schemas *EnterC* and *FindMaxC* describe the concrete operations of entering a number and finding the maximum.

$$
\begin{array}{|l}
\textit{EnterC} \\
\hline
\Delta \mathit{MaxC} \\
\mathit{number}? : \mathbb{Z} \\
\hline
(\mathit{setEmpty} = \mathit{true} \land \\
\mathit{setEmpty}' = \mathit{false} \land \\
\mathit{maxNumber}' = \mathit{number}?) \\
\lor \\
(\mathit{setEmpty} = \mathit{false} \land \\
\mathit{setEmpty}' = \mathit{setEmpty} \land \\
\quad ((\mathit{number}? > \mathit{maxNumber} \land \mathit{maxNumber}' = \mathit{number}?) \\
\quad \lor \\
\quad (\mathit{number}? \leq \mathit{maxNumber} \land \mathit{maxNumber}' = \mathit{maxNumber}))) \\
\hline
\end{array}
$$

The concrete operation *EnterC* checks whether a new number is greater than the current maximum. If so, the new input is retained as the new current maximum.

$$
\begin{array}{|l}
\textit{FindMaxC} \\
\hline
\Xi \mathit{MaxC} \\
\mathit{maximum}! : \mathbb{Z} \\
\hline
\mathit{setEmpty} = \mathit{false} \\
\mathit{maximum}! = \mathit{maxNumber} \\
\hline
\end{array}
$$

The operation of outputting the maximum is simply to report the stored number.

□

The incorporation of implementation details makes a specification awkward as is apparent from comparing *Enter* and *EnterC* of Examples 3.1 and 3.2. The main advantages gained from a data refinement are storage and algorithmic efficiency and the greater ease of implementing the data types in the target programming language.

3.1.3 Retrieve Relations

A *retrieve relation*, also commonly known as *abstraction relation* or *abstraction invariant*, is a schema which formally documents the relationship between the abstract and the concrete states [45]. It contains both the abstract and concrete states and further includes predicates to describe the relation between their state variables.

Example 3.3 A retrieve relation *MaxR* for the abstract and concrete states of Examples 3.1 and 3.2 is given below.

$$
\begin{array}{|l}
\textit{MaxR} \\
\hline
\textit{Max} \\
\textit{MaxC} \\
\hline
\textit{setEmpty} = \textit{true} \Leftrightarrow \textit{numbers} = \{\} \\
\textit{max numbers} = \textit{maxNumber} \\
\hline
\end{array}
$$

The retrieve relation says that the boolean variable *setEmpty* is used to indicate whether the set is empty. It also states that the maximum number in the set is the value stored in concrete variable *maxNumber*.

□

Documenting the retrieve relation is important as it contains the design decisions that are made during data refinement and these decisions allow the abstract to be recovered from the concrete. Using this relation, we can prove that the concrete design satisfies the abstract specification.

3.1.4 Proof Obligations

The proof obligations required to show that a concrete design correctly implements an abstract specification are given in this section. For this, assume that the abstract specification consists of a state schema AS, an initial state schema $InitAS$, and an operation schema AOp, and that the corresponding design contains a state CS, an initial state $InitCS$, and an operation COp. Both of the operations AOp and COp have input $x? : X$ and output $y! : Y$, and the abstract and the concrete specifications are related by the retrieve relation $Retr$.

The proof obligations for data refinement may be divided into three kinds: *initial states*, *applicability* and *correctness*. The proof for initial states needs to be performed only once, while the proofs for applicability and correctness must be performed for each operation. These proof requirements are described below.

<u>Initial States</u>

The implemented system must start in one of the states that are prescribed in the abstract specification; as such, each possible initial concrete state must represent a possible initial abstract state. Symbolically, this is written as

$$\begin{array}{l} \forall\, CS' \bullet \\ \quad InitCS \Rightarrow \exists\, AS' \bullet InitAS \wedge Retr'. \end{array}$$

The dashes are necessary because, by convention, the state variables in *InitCS* and *InitAS* are dashed.

Note that with this requirement, we are allowing fewer concrete initial states than abstract states. This is acceptable because our abstract specification insists only that the system start in *one* of the initial states; as such, we demand only that each concrete initial state represents a legal abstract initial state.

Applicability

An implemented operation must be at least as applicable as its specification. This means that whenever the precondition of the abstract operation is satisfied, the precondition of its concrete version, as related by the retrieve relation, must also be true. Symbolically, this is written as

$$\begin{array}{l}\forall\, AS;\ CS;\ x? : X \bullet \\ \qquad \text{pre } AOp \land Retr \Rightarrow \text{pre } COp.\end{array}$$

Since the precondition of the concrete operation may be more general than the precondition of the abstract operation, the concrete operation may be used in more situations. As such, the concrete operation may be more applicable than its abstract counterpart.

Correctness

Since the precondition of an operation describes when a terminating state is guaranteed, the applicability requirement says that if the abstract operation terminates, its concrete version must also do so. An additional requirement for the concrete operation to be correct is for it to terminate in a state that is agreeable to its abstract specification. Symbolically, this is written as

$$\begin{array}{l} \forall AS;\ CS;\ CS';\ x? : X;\ y! : Y \bullet \\ \quad \text{pre } AOp \land Retr \land COp \Rightarrow (\exists AS' \bullet AOp \land Retr'). \end{array}$$

The condition may be understood as: if the concrete operation were to be invoked under the precondition of its abstract specification, it must produce a result that is within the requirements of its abstract specification.

Example 3.4 The conditions required to prove the satisfiability of the concrete design in Example 3.2 are given below. For the initial states, the required condition is

$$\begin{array}{l} \forall MaxC' \bullet \\ \quad InitMaxC \Rightarrow \exists Max' \bullet InitMax \land MaxR'. \end{array}$$

In order to show the applicability of the concrete operations, we need to show

$$\begin{array}{l} \forall Max;\ MaxC;\ number? : \mathbb{Z} \bullet \\ \quad \text{pre } Enter \land MaxR \Rightarrow \text{pre } EnterC \end{array}$$

and

$$\begin{array}{l} \forall Max;\ MaxC \bullet \\ \quad \text{pre } FindMax \land MaxR \Rightarrow \text{pre } FindMaxC. \end{array}$$

The requirements for the correctness of both the operations are

Operation	Precondition
$Enter$	$true$
$FindMax$	$numbers \neq \{\}$
$EnterC$	$true$
$FindMaxC$	$setEmpty = false$

Table 3.1: The preconditions of the operations $Enter$, $FindMax$, $EnterC$, and $FindMaxC$.

$$\begin{array}{l}\forall\, Max;\ MaxC;\ MaxC';\ number? : \mathbb{Z} \bullet \\ \quad \text{pre } Enter \land MaxR \land EnterC \Rightarrow (\exists\, Max' \bullet Enter \land MaxR')\end{array}$$

and

$$\begin{array}{l}\forall\, Max;\ MaxC;\ MaxC';\ maximum! : \mathbb{Z} \bullet \\ \quad \text{pre } FindMax \land MaxR \land FindMaxC \Rightarrow (\exists\, Max' \bullet FindMax \land MaxR').\end{array}$$

□

Example 3.5 We demonstrate how the proof obligations for the concrete operation $EnterC$ may be discharged. Its precondition may be found in Table 3.1.

Applicability

Since the precondition of $EnterC$ is true, the condition

$$\text{pre } Enter \land MaxR \Rightarrow \text{pre } EnterC$$

is trivially satisfied and the applicability of $EnterC$ is established.

Correctness

To prove correctness of $EnterC$, we need to show that

$$\begin{array}{l} \forall\, Max;\ MaxC;\ MaxC';\ number? : \mathbb{Z} \bullet \\ \quad \text{pre } Enter \land MaxR \land EnterC \Rightarrow (\exists\, Max' \bullet Enter \land MaxR'). \end{array}$$

First, we simplify the consequent of the condition which is

$$\exists\, Max' \bullet Enter \land MaxR'.$$

When expanded, this yields

$$\begin{array}{ll} \Leftrightarrow & \exists\, numbers' : \mathbb{Z} \bullet \\ & \quad numbers' = numbers \cup \{number?\} \land \\ & \quad setEmpty' = true \Leftrightarrow numbers' = \{\} \land \\ & \quad max\ numbers' = maxNumber'. \end{array}$$

Using the one-point rule, we may eliminate $numbers'$ and arrive at

$$\begin{array}{ll} \Leftrightarrow & setEmpty' \neq true \land \\ & max\ (numbers \cup \{number?\}) = maxNumber'. \end{array}$$

This simplified form of the consequent is substituted into the original condition to yield a simpler requirement for correctness, which is as follows.

$$\begin{array}{l} MaxR \land EnterC \Rightarrow \\ \quad setEmpty' \neq true \land max\ (numbers \cup \{number?\}) = maxNumber' \end{array}$$

We have omitted pre $Enter$ from the condition since it is true.

We may now proceed to establish the new correctness requirement. Analyzing the different cases in $EnterC$, the premise of the requirement may be rewritten as the following three disjuncts after a few steps of logical manipulation.

$$\begin{array}{l}
(MaxR \wedge \\
setEmpty = true \wedge \\
setEmpty' = false \wedge \\
maxNumber' = number?) \\
\vee \\
(MaxR \wedge \\
setEmpty = false \wedge \\
setEmpty' = setEmpty \wedge \\
number? > maxNumber \wedge \\
maxNumber' = number?) \\
\vee \\
(MaxR \wedge \\
setEmpty = false \wedge \\
setEmpty' = setEmpty \wedge \\
number? \leq maxNumber \wedge \\
maxNumber' = maxNumber)
\end{array}$$

Separately, each of these disjuncts may be shown to imply the consequent. We show the exercise for only the first. Fully writing out the first disjunct, we get

$$\begin{array}{ll}
\Leftrightarrow & (setEmpty = true \Leftrightarrow numbers = \{\}) \wedge \\
 & max\ numbers = maxNumber \wedge \\
 & setEmpty = true \wedge \\
 & setEmpty' = false \wedge \\
 & maxNumber' = number?.
\end{array}$$

Substituting the definition of $setEmpty$ and leaving out the second conjunct, we have

$$\begin{array}{ll}
\Rightarrow & setEmpty' = false \wedge \\
 & numbers = \{\} \wedge \\
 & maxNumber' = number?.
\end{array}$$

Using the properties of max and sets, we have

$$\begin{aligned}\Rightarrow\ & setEmpty' = false \land \\ & numbers = \{\} \land \\ & maxNumber' = max\ \{number?\}.\end{aligned}$$

Using a property of set, we get

$$\begin{aligned}\Rightarrow\ & setEmpty' = false \land \\ & numbers = \{\} \land \\ & maxNumber' = max\ (\{\} \cup \{number?\}).\end{aligned}$$

Substituting $\{\}$ for *numbers*, we arrive at

$$\begin{aligned}\Rightarrow\ & setEmpty' = false \land \\ & maxNumber' = max\ (numbers \cup \{number?\}).\end{aligned}$$

And, since $true \neq false$, this implies

$$\begin{aligned}\Rightarrow\ & setEmpty' \neq true \land \\ & maxNumber' = max\ (numbers \cup \{number?\})\end{aligned}$$

which is exactly what we need.

$\square$

3.1.5 Proof Obligations for Functional Retrieve Relation

Each concrete state frequently represents exactly one abstract state, and the retrieve relation may be viewed as a total function from concrete states to abstract states. When this happens, the retrieve relation is termed as being *functional.*

Simpler proof obligations may be used when the retrieve relation is functional [42, 45]. The conditions for initial states and correctness are easier to prove although the requirement for applicability remains the same.

Initial States

$$\begin{array}{l}\forall AS';\ CS' \bullet \\ \quad InitCS \land Retr' \Rightarrow InitAS\end{array}$$

Applicability

$$\begin{array}{l}\forall AS;\ CS;\ x? : X \bullet \\ \quad \text{pre } AOp \land Retr \Rightarrow \text{pre } COp\end{array}$$

Correctness

$$\begin{array}{l}\forall AS;\ AS';\ CS;\ CS';\ x? : X;\ y! : Y \bullet \\ \quad \text{pre } AOp \land Retr \land COp \land Retr' \Rightarrow AOp\end{array}$$

The main benefit for using these is that the existential quantifiers may be avoided.

3.1.6 Proving Retrieve Relations to be Functional

In order to show that a retrieve relation is functional we need to prove

$$\forall CS \bullet \exists_1 AS \bullet Retr.$$

As indicated in [45], a sufficient condition for proving that a retrieve relation is functional is to show that there is an equation that defines each abstract component's value in terms of concrete components and total functions.

3.2 Case Study

In the following, we describe the data refinement of the abstract specification of the stack from Chapter 2. This example complements the one in the earlier part of this chapter as it contains error handling and uses schema connectives. For convenience, we assume that the data types used here may be found in the target programming language.

3.2.1 Concrete States

The stack is implemented by using an array of *max* cells, each of which stores an element of type *OBJECT*. An integer variable is also included to keep track of the index of the top element in the stack. This concrete state is described in *StackC*.

$$
\begin{array}{|l}
\hline
\textit{StackC} \\
stackC : 1..max \rightarrow OBJECT \\
topC : \mathbb{Z} \\
\hline
0 \leq topC \leq max \\
\hline
\end{array}
$$

The array in our stack is modeled as a total function whose domain is the set of consecutive integers from 1 to *max*. The index of the top element of the stack is given by *topC* which should contain 0 when the stack is empty.

3.2.2 Retrieve Relation

The next step is to relate the abstract and concrete states. This is done in the schema *StackR*.

```
┌─ StackR ──────────────────────────────
│ Stack
│ StackC
├──────────
│ stack = 1..topC ◁ stackC
└───────────────────────────────────────
```

Using the domain restriction symbol $\lhd$, the expression $1..topC \lhd stackC$ yields a function which is the same as *stackC*, except that it is only valid for the domain $1..topC$. Since a sequence in Z is defined as a function whose domain is a set of consecutive non-zero natural numbers starting at one, the predicate in *StackR* requires the sequence *stack* to have the same elements as the first *topC* cells of array *stackC*.

Note that exactly one value of *stack* may be derived for every value of the concrete components *topC* and *stackC*. Hence, we know from the discussion in Section 3.1.5 that the retrieve relation is functional. As such, the simpler set of proof obligations may be used.

3.2.3 Initial Concrete States

The schema *InitStackC* which describes the initial concrete states requires that the index of the top element be 0.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad InitStackC \quad}} \\
StackC' \\
\hline
topC' = 0 \\
\hline
\end{array}$$

3.2.4 Proof Obligation for Initial State

The proof obligation for the initial state is stated below.

$$\begin{array}{l}
\forall Stack';\ StackC' \bullet \\
\qquad InitStackC \land StackR' \Rightarrow InitStack
\end{array}$$

The proof may be conducted as follows. From $InitStackC \land StackR'$, we know that $topC' = 0 \land stack' = 1..topC' \lhd stackC'$. Substituting 0 for $topC'$ in the equation for $stack'$, we arrive at the value of an empty set for $stack'$. This implies that $stack'$ is an empty sequence and this is exactly the predicate in $InitStack$.

3.2.5 Concrete Operations

As for the abstract specification, the schemas $\Delta StackC$ and $\Xi StackC$ are also defined for the concrete operations.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \Delta StackC \quad}} \\
StackC \\
StackC' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \Xi StackC \quad}} \\
\Delta StackC \\
\hline
stackC = stackC' \\
topC = topC' \\
\hline
\end{array}$$

The concrete operations may be described in a fashion similar to the abstract ones. We may consider the successful operations and error-handling separately.

Successful Operations

The successful operation for pushing an element onto the concrete stack is described in *PushOkC*.

$$
\begin{array}{|l}
\hline
\textit{PushOkC} \\
\Delta \textit{StackC} \\
\textit{object?} : \textit{OBJECT} \\
\hline
\textit{topC} < \textit{max} \\
\textit{topC}' = \textit{topC} + 1 \\
\textit{stackC}' = \textit{stackC} \oplus \{\textit{topC}' \mapsto \textit{object?}\} \\
\hline
\end{array}
$$

The use of the overriding operator $\oplus$ in the last predicate of the schema needs some elaboration. For functions P and Q, $P \oplus Q$ is the relation containing all the ordered pairs of Q, and when the first element of an ordered pair of P does not appear in the domain of Q, that ordered pair is also included. Therefore, $P \oplus Q$ may be viewed as a merge of P and Q, under the condition that when there is a domain conflict, the elements of Q are selected over those of P. Hence, the predicate $\textit{stackC}' = \textit{stackC} \oplus \{\textit{topC}' \mapsto \textit{object?}\}$ says that the array $\textit{stackC}'$ is the same as *stackC* except that the value in the $\textit{topC}'$th cell of $\textit{stackC}'$ is *object?*.

The successful operation for popping an element off the concrete stack is described in *PopOkC*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, PopOkC \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta StackC \\
\hline
topC \neq 0 \\
topC' = topC - 1 \\
stackC' = stackC \\
\hline
\end{array}$$

The concrete stack is popped by decrementing the index of the top element.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, TopOkC \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi StackC \\
object! : OBJECT \\
\hline
topC \neq 0 \\
object! = StackC(topC) \\
\hline
\end{array}$$

The value of the top element is the value of the element of the array with index $topC$.

Error Handling

The concrete error handling operations are defined similar to the abstract ones.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, StackFullC \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi StackC \\
result! : REPORT \\
\hline
topC = max \\
result! = full \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, StackEmptyC \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi StackC \\
result! : REPORT \\
\hline
topC = 0 \\
result! = empty \\
\hline
\end{array}$$

Operation	Precondition
$PushOkC$	$topC < max$
$PopOkC$	$topC \neq 0$
$TopOkC$	$topC \neq 0$
$StackFullC$	$topC = max$
$StackEmptyC$	$topC = 0$
$PopC$	$true$
$PushC$	$true$
$TopC$	$true$

Table 3.2: The preconditions of the concrete operations of the stack.

The successful and error handling operations are combined as in the abstract specification.

$$\begin{array}{l} PopC \mathrel{\widehat{=}} (PopOkC \land Success) \lor StackEmptyC \\ PushC \mathrel{\widehat{=}} (PushOkC \land Success) \lor StackFullC \\ TopC \mathrel{\widehat{=}} (TopOkC \land Success) \lor StackEmptyC \end{array}$$

As the reader will notice in later sections of this chapter, combining the concrete operations in a way similar to the combination of the abstract operations enables the proof obligations of data refinement to be organized based on the structure of the operations. The preconditions of the concrete operations are given in Table 3.2. Notice that the concrete versions of operations, $PopC$, $PushC$, and $TopC$, are also total operations.

3.2.6 Proof Obligations for Concrete Operations

The conditions for showing the applicability and correctness of $PushC$, $PopC$, and $TopC$ are given below. Since the retrieve relation $StackR$ is functional, the conditions for functional refinement are used.

Applicability

$$\forall\, Stack;\ StackC;\ object? : OBJECT \bullet$$
$$\quad \text{pre } Push \land StackR \Rightarrow \text{pre } PushC$$

$$\forall\, Stack;\ StackC \bullet$$
$$\quad \text{pre } Pop \land StackR \Rightarrow \text{pre } PopC$$

$$\forall\, Stack;\ StackC \bullet$$
$$\quad \text{pre } Top \land StackR \Rightarrow \text{pre } TopC$$

Recall that the preconditions of these abstract and concrete operations are all true. As such, the consequents of the implications are all true and hence, these conditions are trivially satisfied.

Correctness

$$\forall\, Stack;\ Stack';\ StackC;\ StackC';$$
$$\qquad object? : OBJECT;\ result! : REPORT \bullet$$
$$\quad \text{pre } Push \land StackR \land PushC \land StackR' \Rightarrow Push$$

$$\forall\, Stack;\ Stack';\ StackC;\ StackC';$$
$$\qquad report! : REPORT \bullet$$
$$\quad \text{pre } Pop \land StackR \land PopC \land StackR' \Rightarrow Pop$$

$$\forall\, Stack;\ Stack';\ StackC;\ StackC';$$
$$\qquad report! : REPORT;\ object! : OBJECT \bullet$$
$$\quad \text{pre } Top \land StackR \land TopC \land StackR' \Rightarrow Top$$

Each of these may be proved by considering the successful and error-handling parts separately. To illustrate this process, the steps for proving the correctness of $PushC$ are given in the following example.

Example 3.6 This example shows how the correctness of *PushC* may be proved. The premise of the correctness condition for *PushC* is

$$\text{pre } \mathit{Push} \wedge \mathit{StackR} \wedge \mathit{PushC} \wedge \mathit{StackR}'.$$

By absorbing pre *Push* (since it is true) and substituting $((\mathit{PushOkC} \wedge \mathit{Success}) \vee \mathit{StackFullC})$ for *PushC*, and after some logical manipulation, we arrive at

$$\begin{array}{l}(\mathit{StackR} \wedge \mathit{StackR}' \wedge (\mathit{PushOkC} \wedge \mathit{Success}))\\ \vee\\ (\mathit{StackR} \wedge \mathit{StackR}' \wedge \mathit{StackFullC}).\end{array}$$

Since $\mathit{Push} \mathrel{\widehat{=}} (\mathit{PushOk} \wedge \mathit{Success}) \vee \mathit{StackFull}$, a strategy would be to divide the proof into success and error parts, thus structuring the proof based on the way the schemas are connected logically. Hence, we aim to prove

$$(\mathit{StackR} \wedge \mathit{StackR}' \wedge (\mathit{PushOkC} \wedge \mathit{Success})) \Rightarrow (\mathit{PushOk} \wedge \mathit{Success})$$

and

$$(\mathit{StackR} \wedge \mathit{StackR}' \wedge \mathit{StackFullC}) \Rightarrow \mathit{StackFull}$$

separately to complete the proof. We show below this process for the success part. Expanding

$$\mathit{StackR} \wedge \mathit{StackR}' \wedge (\mathit{PushOkC} \wedge \mathit{Success}),$$

we get

$$\begin{aligned}
\Leftrightarrow\ & stack = 1..topC \lhd stackC \land \\
& stack' = 1..topC'' \lhd stackC'' \land \\
& topC < max \land \\
& topC'' = topC + 1 \land \\
& stackC'' = stackC \oplus \{topC'' \mapsto object?\} \land \\
& result! = ok.
\end{aligned}$$

Substituting $topC''$ and $stackC'$ with their definitions, we get

$$\begin{aligned}
\Rightarrow\ & stack = 1..topC \lhd stackC \land \\
& stack' = (1..topC + 1) \lhd (stackC \oplus \{topC + 1 \mapsto object?\}) \land \\
& topC < max \land \\
& result! = ok.
\end{aligned}$$

Using a property of domain restriction $\lhd$, and realizing that the domain of $stackC$ is $1..max$, we deduce

$$\begin{aligned}
\Rightarrow\ & stack = 1..topC \lhd stackC \land \\
& stack' = (1..topC) \lhd stackC \cup \{topC + 1 \mapsto object?\} \land \\
& topC < max \land \\
& result! = ok.
\end{aligned}$$

Using the relationship between functions and sequences, we arrive at

$$\begin{aligned}
\Rightarrow\ & stack = 1..topC \lhd stackC \land \\
& stack' = stack \frown \langle object? \rangle \land \\
& topC < max \land \\
& result! = ok.
\end{aligned}$$

Since the cardinality of a function can never be greater than that of its domain, we have

$$\begin{aligned}
\Rightarrow\ & \# stack \leq topC \land \\
& stack' = stack \frown \langle object? \rangle \land \\
& topC < max \land \\
& result! = ok.
\end{aligned}$$

Since $topC < max$, we deduce

$$\begin{aligned} \Rightarrow\ & \#stack < max \land \\ & stack' = stack \frown \langle object? \rangle \land \\ & result! = ok. \end{aligned}$$

which is exactly $(PushOk \land Success)$.

□

3.3 Summary and Bibliographical Notes

In this chapter, we presented a method of data refinement. This involves proposing a concrete design containing the concrete state space and operations, and proving that this design satisfies its abstract specification. Using examples, we have shown how the concrete operations may be proposed so that they are structurally similar to their abstract counterparts with respect to logical schema connectives. We further indicate how the proof obligations arising from the refinement may be discharged while exploiting this structural similarity.

In our account, we have given an ideal situation where a concrete design may be produced from an abstract specification in just one refinement step. In many cases, especially for complex and large systems, it may be necessary to go through a series of refinement steps that produce a number of intermediate designs, each of which contains more implementation detail than those previous. The final design which is then accepted as the concrete design should contain data types that are storage and algorithmic efficient, and are easily constructed in the target programming language.

Our primary references for data refinement within the framework of Z are [42, 45] and the use of this technique may be observed in [45, 24, 25, 42]. The interested reader may find in [20] a theoretical investigation of refinement within the Z framework.

There exists a complementary technique where a concrete operation may be *calculated* directly from its abstract specification and the retrieve relation. Theoretical work concerning this calculative mode of data refinement may be found in [22, 21] and examples of its use may be found in [22, 45].

Chapter 4

Translation into the Refinement Calculus

A concrete design is the specification of a software system containing data types which can be easily realized as data structures in the target programming language. This chapter and the next chapters show how a program that implements the software system may be calculated from its concrete design using a formal development method called the *refinement calculus* [31]. Since the notation of the refinement calculus is different from that of Z, the concrete design must first be translated into the refinement calculus before the calculus may be applied.

In this chapter, we concern ourselves with the issues arising from the translation from Z to the refinement calculus. A brief introduction to the refinement calculus is given so that the reader may appreciate the necessity of and strategies for this translation.

4.1 The Notation of the Refinement Calculus

To provide the notational requirements for program development, the refinement calculus contains a language that may be used to describe both specifications and programs in the same framework. This is achieved by employing both non-executable and executable constructs.

Non-executable constructs are used mainly for specification, while the executable constructs represent (executable) programs. The only non-executable construct is a *specification statement*. The executable constructs are drawn mainly from Dijkstra's language of guarded commands, and include *assignment*, *alternation*, *iteration*, and *sequential composition*.

4.1.1 Specification Statements

A *specification statement* has the form

$$w : [pre\ ,\ post].$$

The term w is called the *frame* and is used to represent a possibly empty list of variables. The predicates pre and $post$ are the pre- and postconditions describing before- and after-states. This construct may be used to specify a program that, by changing only the variables in w, brings the state of a system from one that satisfies *pre* to one that satisfies *post*.

<u>Initial Variables</u>

In the refinement calculus, the before- and after-values of a variable are distinguished by representing the before-value of a variable with that variable subscripted with a zero. We call zero-subscripted variables *initial variables* and they are allowed only in the postconditions of specification statements.

Example 4.1 Assuming that x and y are integer variables, the specification statement

$$x, y : [x \geq 0 \;,\; y > x_0]$$

describes a program that has the before- and after-states described by $x \geq 0$ and $y > x_0$ respectively. Since x_0 refers to the before-value of x, the execution of the program must give the variable y a value greater than the original value of x. The program may change the value of x if it wishes. □

4.1.2 Assignments

A *single assignment* has the form

$$w := E.$$

When this is executed, the variable w takes on the value given by the expression E. The language also provides a *multiple assignment* which has the form

$$w_1, ..., w_n := E_1, ..., E_n.$$

When this is executed, each E_i is simultaneously assigned to its corresponding w_i, for $i \leq i \leq n$.

4.1.3 Alternations

An *alternation* may be used to implement case analysis. It has the form

```
if  $G_1 \rightarrow prog_1$
[]  $G_2 \rightarrow prog_2$
     ⋮
[]  $G_n \rightarrow prog_n$
fi
```

and may also be written as the generalized

$$\textbf{if } ([] \;\; i \bullet G_i \rightarrow prog_i) \textbf{ fi}.$$

Each $G_i \rightarrow prog_i$ is called a guarded command, and each predicate G_i is known as a *guard* and each program $prog_i$ is known as a *command*. When this construct is executed, all of the guards are evaluated. If exactly one of these guards is true, its command is executed. If more than one of these guards are true, *any one* of the commands associated with these guards is executed. If none of these guards is true, the behavior of the alternation is undefined. In other words, failure to satisfy at least one of the guards should be regarded as disastrous.

To elaborate on this last point, note that a single guard alternation is similar to a conventional conditional statement without its else part. If the conditional statement is executed when its condition is false, then its execution yields no effect. However, an execution of the alternation when the guard is false will cause its behavior to be indeterminate.

4.1.4 Iterations

An *iteration* may be used to implement repetition. It has the form

$$\begin{array}{ll} \textbf{do} & G_1 \rightarrow prog_1 \\ [] & G_2 \rightarrow prog_2 \\ & \vdots \\ [] & G_n \rightarrow prog_n \\ \textbf{od} & \end{array}$$

and may also be written as the generalized

$$\textbf{do}\ ([]\ i \bullet G_i \rightarrow prog_i)\ \textbf{od}.$$

When this is executed, all the guards are evaluated and the command that is associated with one of the true guards is executed. This is repeated until no guard is true which then causes the iteration to terminate successfully.

4.1.5 Sequential Compositions

A *sequential composition* has the form

$$P\ ;\ Q.$$

This allows a larger program $P\ ;\ Q$ to be built from smaller programs P and Q. When this construct is executed, the program P is first executed, followed by Q.

```
|[
    var $x : \mathbb{Z}$;
    and $x > 0$;
    procedure $Proc$(value result $a : \mathbb{Z}$) $\triangleq$
        |[
            ⋮
        ]|
    •
        ⋮
    $Proc(x)$;
        ⋮
]|
```

Figure 4.1: The skeleton of a sample program.

4.1.6 Local Blocks, Variables, Invariants, and Procedures

The notation of the refinement calculus allows *variables*, *invariants*, *local blocks*, and *procedures* to be declared. Examples of these may be found in the program skeleton of Figure 4.1.

Local Blocks

A *block* has the general form

$$[\![\; Declaration \; \bullet \; Body \;]\!]$$

and is delimited by the symbols $[\![$ and $]\!]$. The *Declaration* part of the block contains the declarations of variables, invariants, and procedures, while the *Body*

```
|[
    var $a : \mathbb{Z}$;
    ⋮
        |[
            var $a : \mathbb{Z}$
            ⋮
            $B$
            ⋮
        ]|
    ⋮
    $A$
    ⋮
]|
```

Figure 4.2: Nested blocks.

part contains a program made up of constructs like specification statements, assignments, iterations, etc.

Variables

Variable declarations must be done immediately after the |[symbol of a block. Variables are declared by preceding them with the keyword var and giving their names and types. For illustration, the integer variable x is declared in the program of Figure 4.1.

The *scope* of a variable is the block in which it is declared. When blocks are *nested*, a variable in an inner block hides the outer block variable with the same name. For example, in Figure 4.2, the inclusion of variable a at point B refers to the value of a of the inner block. On the other hand, the a at point A refers to the value of a in the outer block.

Invariants

The *invariant* of a variable may be specified with the keyword and immediately after the variable's declaration. In Figure 4.1, the variable x has an invariant saying that it must always be positive.

Procedures

A *procedure* may be declared with the keyword **procedure**. This declaration gives the procedure's name and its *formal parameters* (optional). The *text* of the procedure, which is usually in a local block, is separated from the name and the formal parameters by the symbol $\triangleq$.

The *call-by-value*, *call-by-result*, and *call-by-value-result* substitution methods for passing parameters are available. In Figure 4.1, a procedure called *Proc* is declared, which has a call-by-value-result formal parameter a.

A procedure may be called within the local block for which it is declared by including its name and any *actual parameters*. A call to procedure *Proc* is included in the body of the program of Figure 4.1.

```
$[\![$ var $x, y : \mathbb{Z} \bullet$
    $x, y : [true, (x_0 \geq y_0 \wedge x = y_0 \wedge y = x_0) \vee$
                  $(y_0 \geq x_0 \wedge x = x_0 \wedge y = y_0)]$
$]\!]$
```

Figure 4.3: An abstract program.

4.2 Using the Refinement Calculus

The refinement calculus provides a notation and a large collection of laws for program development. A *program*, in the refinement calculus, refers to a piece of text which is made up of the executable and non-executable constructs. A program that is to be developed is specified in terms of specification statements and these are gradually transformed using these laws to yield only executable constructs. This transformation, known as refinement, is explained in greater detail in Section 4.2.4.

4.2.1 Abstract Programs

An *abstract program* is one that contains at least one specification statement within its body. A program is also known as an *abstract* program since it may contain specification statements. An example of such a program may be found in Figure 4.3. The specification in this program requires that the values of x and y be swapped so that $y \geq x$ after its execution.

```
|[
    var $x, y : \mathbb{Z}$;
    procedure $Swap \triangleq$
        |[ var $z : \mathbb{Z}$ •
              $z := x$;
              $x := y$;
              $y := z$
        ]|
    •
    if  $x \geq y \rightarrow Swap$
    []  $y \geq x \rightarrow$ skip
    fi
]|
```

Figure 4.4: An executable program.

4.2.2 Executable Programs

An *executable program* is one that contains only executable constructs. An executable program which implements the abstract program of Figure 4.3 may be found in Figure 4.4.

4.2.3 A Liberal View of Programs

The word *program* is used loosely in the world of the refinement calculus. In addition to the conventional view that a program contains only executable constructs, a program here can also mean an abstract program with only specification statements, which is regarded as only a specification. Programs may also contain single (or multiple) non-executable and executable constructs. A specification statement, iteration, and alternation are all examples of *atomic* programs. The programs formed by sequentially composing atomic programs are known as *compound* programs.

The liberal use of the term *program* offers a convenience: we are relieved of the burden of describing seemingly similar things with different terms, thereby allowing us to concentrate on the mathematical requirements of program development. All this being said, it is still important to reserve the term *specification* for a program composed only of specifications statements, and *code* for a program composed only of executable constructs.

4.2.4 Refinement

For programs P and Q,

$$P \sqsubseteq Q,$$

(pronounced Q refines P) means that Q is a *better* program than P. For instance, P may be a specification statement and Q may be some code that implements P. When this refinement step is performed using the laws of the refinement calculus, Q is guaranteed to satisfy P.

The refinement calculus may be used in the development of a software system. After specifying the software system as an abstract program, an executable program may be calculated from the abstract through a series of refinement steps.

```
|[
    var $x, y : \mathbb{Z}$;
    procedure $Swap \mathrel{\widehat=}$
        |[
            var $z : \mathbb{Z}$ •
            $x, y : [true, x = y_0 \wedge y = x_0]$
        ]|
    •
    if  $x \geq y \rightarrow Swap$
    []  $y \geq x \rightarrow$ skip
    fi
]|
```

Figure 4.5: An abstract program containing both specification statements and executable constructs.

Each refinement step introduces more executable constructs until all the specification statements are refined into code. Assuming that the original specification is S and the finished code is C, this refinement may be written as

$$S \sqsubseteq M_1 \sqsubseteq \cdots \sqsubseteq M_i \sqsubseteq \cdots \sqsubseteq C$$

where each of the intermediate M_i is an abstract program containing both specification statements and executable constructs. For example, the program in Figure 4.5 may be an intermediate program created along the refinement of the program of Figure 4.3 into the program of Figure 4.4.

4.2.5 Some Simple Laws

In this section, we give some simple laws of refinement and examples of their use. This should provide the reader with an indication of what a typical refinement step looks like.

Law 4.1 (weaken precondition "wp") If $pre \Rightarrow pre'$, then

$$w : [pre\ ,\ post] \sqsubseteq w : [pre'\ ,\ post].$$

□

Law "wp" says that a program may be refined into one that is more applicable. Since pre' is more general than pre, the refined program may be used more generally.

Example 4.2 Since $x \geq 0 \Rightarrow true$,

$$\begin{array}{ll} & y : [x \geq 0\ ,\ y > x_0] \\ \sqsubseteq & \text{"wp"} \\ & y : [true\ ,\ y > x_0]. \end{array}$$

The result of the refinement is a program that is applicable in all circumstances, rather than one that is applicable for only $x \geq 0$.

□

Law 4.2 (strengthen postcondition "sp") If $pre[w \backslash w_0] \wedge post' \Rightarrow post$, then

$$w : [pre\ ,\ post] \Rightarrow w : [pre\ ,\ post'].$$

□

Law "sp" says that a program may be refined into one that is more definite. Since $post' \Rightarrow post$, a program that terminates in a state described by $post'$ also terminates in a state described by $post$. What we gain from the refinement is the additional information provided by $post'$, since $post'$ is stronger than $post$.

Example 4.3 Since $y = x_0 + 1 \Rightarrow y > x_0$,

$$\begin{array}{ll} & y : [true \ , \ y > x_0] \\ \sqsubseteq & \text{"sp"} \\ & y : [true \ , \ y = x_0 + 1]. \end{array}$$

The only requirement of $y > x_0$ is that y takes on a value greater than the initial value of x. The refinement simply fixes a value for y.

□

Law 4.3 (expand frame "efI")

$$w : [pre \ , \ post] = w, x : [pre \ , \ post \wedge x = x_0].$$

□

Law "efI" says that a specification statement that does not have a variable x in its frame is equivalent to the same specification with x added to its frame and a constraint added to its postcondition saying that x does not change. Note that an equality between the two specification statements is used to indicate that the refinement may go both ways.

Example 4.4

$$
\begin{aligned}
& y : [x \geq 0 \ , \ y > x_0] \\
= \ & x, y : [x \geq 0 \ , \ y > x_0 \wedge x_0 = x]
\end{aligned}
$$

□

4.3 Comparing the Notations of Z and the Refinement Calculus

A comparison of the basis languages of Z and the refinement calculus is given by King [25]. He shows the suitability of the notations for their respective purposes and indicates the necessity of translating from Z to the refinement calculus for program development. His discussion is summarized below.

4.3.1 States

In Z, a state of a simple system may have the form

```
┌─ State ──────────────────────────────
│ v : T
├──────────
│ inv
└──────────────────────────────────────
```

where v is the state variable constrained under the invariant *inv*. In the refinement calculus, the same state variable and system invariant are declared with the keywords var and and respectively:

```
|[
    var $v : T$;
    and $inv$
        ⋮
]|
```

As such, we see a direct correspondence between the two state specifications.

4.3.2 Operations

In Z, an operation with one input and one output may be specified as

```
┌─ $Op$ ──────────────────────────────
│ $\Delta State$
│ $x? : X$
│ $y! : Y$
├──────────
│ $Pred$
└─────────────────────────────────────
```

In the refinement calculus, an operation is specified in terms of a specification statement

$$w : [pre\ ,\ post].$$

As one can see, the operation specifications in Z and the refinement calculus differ in two ways: (i) the schema uses one predicate while the specification statement uses two, and (ii) the specification statement uses the frame while the schema does not.

Single Versus Double Predicates

For the specification of operations, it is more convenient to use only one predicate to relate the before- and after-states. As this predicate incorporates both the pre- and postconditions, it allows operations to be combined by simply performing elementary logic operations such as conjunctions and disjunctions on their predicates. It is the use of only one predicate that enables the powerful features of the schema calculus which are so useful for structuring specifications to be easily applied.

For refinement, it is more convenient to work with a pair of predicates where one of them is the precondition of the operation. The advantage of having the precondition explicit may be seen from the following simple rule of operational refinement using schemas [42]. Assuming that P and Q are schemas describing operations on the state space $State$ with input $x? : X$ and output $y! : Y$. In order to prove $P \sqsubseteq Q$, we need to show

$$\begin{array}{l} \forall\, State;\ x? : X \bullet \\ \qquad \text{pre } P \Rightarrow \text{pre } Q \end{array}$$

and

$$\begin{array}{l} \forall\, State;\ State';\ x? : X;\ y! : Y \bullet \\ \qquad \text{pre } P \land Q \Rightarrow P. \end{array}$$

Since such refinements may be performed at several levels, working with preconditions directly will save us the effort of having to calculate them at each level.

Frames

The other major difference between a schema and a specification statement is the presence of a frame. The refinement of a specification statement often gives rise to several specification statements, each indicating the possible change of only a small number of variables. Without the frame, each of the unchanged variables would have to be involved in the postcondition of each of the specifications. Such specifications would become excessively complex and unmanageable. With the frame, a variable may be specified as unchanged simply by leaving it out of the frame. The use of the frame relieves the developer of the burden of writing $x = x_0$ for each unchanged variable x.

4.3.3 Before- and After-State Variables

Z and the refinement calculus differ also in the way before- and after-state variables are distinguished. In Z, the undashed name of a variable, say x, would refer to its value in the before-state, while the dashed version, x', would refer to its value in the after-state. For a variable in the refinement calculus, its undecorated name, x, would refer to its value in the after state, while the zero-subscripted version, x_0, would refer to its value in the before-state. This distinction is made only in the postcondition of a specification since the precondition always refer to before-state values. Since a postcondition is used to specify after-states, it is more common for it to refer to after-state variables rather than before-state ones.

Furthermore, the proper use of the frame would have alleviated the need to write $x = x_0$ for each unchanged variable x, which again indicates that the before-state variables appear less frequently. As such, it is more economical and simpler to decorate the before-state variables.

4.3.4 Renaming Versus Substitution

In Z, the schema expression

$$S[y/x]$$

for schema S with component y would mean the same schema with all the occurrences of y replaced by x. This is the commonly used operation called *schema renaming*. In the refinement calculus, there is a similar notion called *substitution*. For a predicate P,

$$P[x \backslash y]$$

obtains P with free occurrences of the variable x replaced by the term y.

Woodcock has suggested using the symbol / for substitution in the refinement calculus [45]. We have decided not to use this as the original notation is more elegant for the refinement of procedures, as will be shown later. Instead, we have chosen to use the symbol \ for schema renaming. Although this symbol is used also as the schema *hiding* operator, there should be no confusion since the renaming operator occurs in square brackets ([ ]) while the hiding operator does not.

4.4 Rules for Change of Notation

The discussion in the preceding sections examined the considerations that arise when translating from Z to the refinement calculus. Rules for translation based on these considerations are first worked out by King [25]. We use the version that is presented by Woodcock in [45] since this version is more intuitive.

4.4.1 Basic Rules

The Rule "cc" concerns the convention for distinguishing before- and after-state variables.

Rule 4.1 (change conventions "cc") Let Op be a schema and $[\![Op]\!]$ denote the same schema with the convention changed to that of the refinement calculus. If Op has state variables v, then

$$[\![Op]\!] \mathrel{\widehat{=}} Op[v_0, v \backslash v, v'].$$

□

The Rule "sss" concerns the the translation of states and operations.

Rule 4.2 (schema to specification statements "sss") Let Op be a schema describing an operation with input $x?$ and output $y!$ on a state $State$ which contains variables v:

State

$v : T$

inv

Op

$\Delta State$

$x? : X$

$y! : Y$

Pred

The description of the state translates into the following declaration:

var $v : T$
and *inv*.

The operation translates into the following specification statement:

$v, y! : [\text{pre } Op \ , \ [\![Op]\!]]$

Notice that the schemas are used as predicates in this specification statement. When this happens, these predicates refer to the predicate part of the schemas.

□

4.4.2 Specifications to Abstract Programs

Using the rules "cc" and "sss", a Z specification may be translated into an abstract program. This process is illustrated in the following example.

Variable	Abbreviation
$maxNumber$	mN
$setEmpty$	sE
$number?$	n
$maximum!$	m

Table 4.1: Abbreviations for the state, input and output variables of Example 3.2.

Example 4.5 Here, the concrete design of Example 3.2 is translated into an abstract program. A convention of using the refinement is to have short variable names because they will be copied quite frequently during refinement. We abbreviate the state, input and output variable names as shown in Table 4.1.

The states and operations are translated according to Rule "sss". Furthermore, each operation is transformed into a procedure. The resultant program may be found in Figure 4.6. □

The only remaining issue is the design of the main program which uses these procedures. In Example 4.5, this program is *mainProg* and its content is the subject of the next section.

Main Programs

The *main program* is one that initializes the system and uses the procedures to perform the functions of the system. The main program may be written as

$$initProg \ ; \ prog$$

```
|[
    var $mN : \mathbb{Z}$
    $sE : BOOLEAN$

    procedure $InitMaxC \triangleq$
        $mN, sE : [true\ ,\ sE]$

    procedure $EnterC$(value $n : \mathbb{Z}$) $\triangleq$
        $mN, sE : [true\ ,\ (sE_0 \wedge \neg sE \wedge mN = n)$
                              $\vee$
                              $(\neg sE_0 \wedge sE = sE_0 \wedge$
                                          $((n_0 > mN_0 \wedge mN = n)$
                                          $\vee$
                                          $(n_0 \leq mN_0 \wedge mN = mN_0)))]$

    procedure $FindMaxC$(result $m : \mathbb{Z}$) $\triangleq$
        $mN, sE, m : [true\ ,\ \neg sE \wedge m = mN_0 \wedge mN = mN_0 \wedge sE = sE_0]$

    •

    $mainProg$
]|
```

Figure 4.6: An abstract program translated from the concrete design of Example 3.2.

where *initProg* is the procedure implementing the initial states and *prog* is the program that uses procedures to perform the functions of the system.

Woodcock describes a popular way of designing *prog* [45]. This involves using a pair of symbols, α and β, to represent the input and output streams. For example, assuming that α and β are both declared as sequences of integers, *mainProg*, the program in Figure 4.6, may be written as

$$InitMaxC\ ;\ \ mN, sE, \alpha, \beta : [true\ ,\ \beta = \langle max\ (\mathrm{ran}\ \alpha)\rangle].$$

This program may then be refined to use the procedures in the abstract program of Figure 4.6.

4.4.3 Simplifying Specification Statements

After a Z operation schema is translated into the refinement calculus, there are often opportunities to simplify the resultant specification statement before any algorithmic refinement is performed. Two simple strategies for such simplification are given below.

Shorten Frame

For a Z operation schema, the predicate contains for each unchanged variable v a constraint of $v = v'$. When this is translated into a specification statement, the postcondition contains $v_0 = v$, with v appearing in its frame. These may be removed by using Law "efI".

Simplifying the Postcondition

Since it is recommended that a Z operation schema contains its precondition explicitly, the specification statement yielded from such a schema will have the precondition restated in its postcondition. Using Law "sp", the precondition may be removed from the postcondition of the specification statement.

4.4.4 Some Derived Rules

Operation schemas often occur as

$$Op \mathrel{\widehat{=}} Op_1 \vee \cdots \vee Op_n$$

or

$$Op \mathrel{\widehat{=}} Op_1 \wedge \cdots \wedge Op_n.$$

In the following, we give rules to translate these schemas directly into abstract programs with some executable constructs. Our rules are generalizations of those found in [25] which are applicable for the case $n = 2$. These derived rules may be shown to be correct refinements with respect to the basic rules of translation of Section 4.4.1. The proofs are omitted here since they are easy.

Rule 4.3 (Alternation Introduction "aiI") Suppose we have

$$Op \mathrel{\widehat{=}} Op_1 \vee \cdots \vee Op_n.$$

If the preconditions of Op_i, $1 \leq i \leq n$, can be expressed in the target programming language, we can translate Op to the following alternation.

```
if   pre $Op_1 \rightarrow Op_1^*$
     ⋮
[]   pre $Op_n \rightarrow Op_n^*$
fi
```

where Op_i^* are the specification statements which result from the use of the Rule "sss".

□

Rule 4.4 (Alternation Introduction "aiII") Suppose we have

$$Op \triangleq Op_1 \vee \cdots \vee Op_n,$$

where pre Op_i, $1 \leq i \leq k \leq n$, is a complex expression that cannot be directly computed in the target programming language. Then, we can translate Op to the following program.

```
|[
    var $b1, ..., bk : BOOLEAN$
    •
    $b1 : [true\ ,\ b1 \Leftrightarrow$ pre $Op_1]$;
    ⋮
    $bk : [true\ ,\ bk \Leftrightarrow$ pre $Op_k]$;
    if   $b1 \rightarrow Op_1^*$
         ⋮
    []   $bk \rightarrow Op_k^*$
    []   pre $Op_{k+1} \rightarrow Op_{k+1}^*$
         ⋮
    []   pre $Op_n \rightarrow Op_n^*$
    fi
]|
```

where $b1, ..., bk$ are fresh variables with scope delimited by $|[$ and $]|$, and Op_i^- are the specification statements which result from the use of the Rule "sss"'. Clearly, for $k = 1$ and $n = 2$, if pre $Op_2 = \neg$pre Op_1, then the second guard may be simplified to $\neg b1$.

□

An application of Rule "aiII" may be found in the next example.

Example 4.6 The concrete design of a simple system which maintains an integer array is given below.

$$\begin{array}{|l} max : \mathbb{Z} \end{array}$$

$$\begin{array}{|l} \textit{State} \\ \hline array : (1..max) \rightarrow \mathbb{Z} \\ \hline \end{array}$$

One of the features of this system is its ability to check whether an input integer is present in the array and to output appropriate messages indicating the presence of this input. This operation is described below as *Find*.

$$REPORT ::= found \mid notFound$$

$$\begin{array}{|l} \textit{Found} \\ \hline \Xi State \\ x? : \mathbb{Z} \\ report! : REPORT \\ \hline \exists k : 1..max \bullet array(k) = x \\ report! = found \\ \hline \end{array}$$

NotFound

$\Xi State$
$x? : \mathbb{Z}$
$report! : REPORT$

$\forall k : 1..max \bullet array(k) \neq x$
$report! = notFound$

$Find \mathrel{\hat=} Found \lor NotFound$

Using Rule "aiII", *Find* may be immediately translated into the following program.

```
|[ var $b : Boolean$ •
     $b : [true\ ,\ b \Leftrightarrow \exists\, k : 1..max \bullet array(k) = x]$
     if   $b \rightarrow$
               $report : [\exists\, k : 1..max \bullet array(k) = x\ ,\ report! = found]$
     []   $\neg b \rightarrow$
               $report : [\forall\, k : 1..max \bullet array(k) \neq x\ ,\ report! = notFound]$
     fi.
]|
```

□

The next rule is the most general of all the translational rules for schema disjunction and it is also the most complex. The reader may find it necessary to read the example that follows in order to understand the rule and appreciate its use. Rules "aiI" and "aiII" may be easily refined from this rule.

Rule 4.5 (Alternation Introduction "aiIII") If we are given

$Op \mathrel{\hat=} Op_1 \lor \cdots \lor Op_n,$

then we can translate Op to the program

```
|[
    var $r : T$
    $r : [true\ ,\ \phi]$;
    if  $\psi_1 \rightarrow \omega : [\phi \wedge \psi_1\ ,\ [\![Op_1]\!]]$
         ⋮
    []  $\psi_n \rightarrow \omega : [\phi \wedge \psi_n\ ,\ [\![Op_n]\!]]$
    fi
]|
```

where ϕ and ψ_i, for $1 \leq i \leq n$, are any predicates, which satisfy the following side conditions.

1. $\phi \wedge (\bigvee i \bullet \text{pre } Op_i) \Rightarrow (\bigvee i \bullet \psi_i)$

2. $\phi \wedge (\bigvee i \bullet \text{pre } Op_i) \Rightarrow (\psi_i \Rightarrow \text{pre } Op_i)$ for $1 \leq i \leq n$.

Notice that if $(\bigvee i \bullet \text{pre } Op_i) = true$, the premises above simplify to ϕ, leaving

1'. $\phi \Rightarrow (\bigvee i \bullet \psi_i)$

2'. $\phi \Rightarrow (\psi_i \Rightarrow \text{pre } Op_i)$ for $1 \leq i \leq n$.

□

An application of Rule "aiIII" may be found in the next example.

Example 4.7 The *Find* operation from Example 4.6 may also be translated using Rule "aiIII".

We intend to have a loop to check the array for an input value. The loop will use an integer variable w to hold the index of the cell that is currently being checked. The loop will step through the array until the integer is found or all the cells are checked. If the integer is found, the loop exits and the value in w will be the index containing the desired integer. Otherwise, w will exceed the index range of the array. Using this strategy, we formulate the predicate ϕ which is designated as H below.

$$H \mathrel{\hat{=}} (w = max + 1 \land x \notin array[1..max]) \lor (w \in 1..max \land array(w) = x)$$

The predicates ψ_1 and ψ_2 may be easily designed as $w \in 1..max$ and $w = max + 1$, and the desired program is obtained according to Rule "aiIII".

```
|[
     var $w : \mathbb{Z}$
     and $1 \leq w \leq max$ •
     $w : [true\ ,\ H]$;
     if   $w \in 1..max \rightarrow$
               $report : [H \land w \in 1..max\ ,\ report! = found]$
     []   $w = max + 1 \rightarrow$
               $report : [H \land w = max + 1\ ,\ report! = notFound]$
     fi.
]|
```

The remaining requirement is to check side conditions. Since *Find* is a total operation, we may use the conditions 1' and 2'. Condition 1' may be expressed as

$$H \Rightarrow ((w \in 1..max) \lor (w = max + 1))$$

which is trivially true. Condition 2' consists of the two subconditions

$$H \Rightarrow (w \in 1..max \Rightarrow \exists\, k : 1..max \bullet array(k) = x)$$

and

$$H \Rightarrow (w = max + 1 \Rightarrow \forall k : 1..max \bullet array(k) \neq x).$$

The proof for the first subcondition may be conducted by assuming $H \wedge w \in 1..max$, and showing that $\exists k : 1..max \bullet array(k) = x$. The second subcondition may also be shown in a similar manner.

□

The following is a derived rule for translating schema conjunctions.

Rule 4.6 (Sequential Composition Introduction "sci") Suppose we have

$$Op \mathrel{\widehat{=}} Op_1 \wedge \cdots \wedge Op_n$$

where Op_i, $1 \leq i \leq n$, takes the form

$$Op_i \mathrel{\widehat{=}} [\Delta State \mid P_i(s_i, s_i')]$$

where s_i are disjoint (vectors of) state variables, and P_i are predicates showing how part of the state is altered. Then Op may be translated into the following program.

$$\begin{array}{l} s_1 : [\text{pre } Op_1, [\![Op_1]\!]]; \\ \vdots \\ s_n : [\text{pre } Op_n, [\![Op_n]\!]]; \end{array}$$

□

Variable	Abbreviation
$stackC$	s
$topC$	t
$object?$	$objI$
$object!$	$objO$
$report!$	$repO$

Table 4.2: Abbreviations for the state, input and output variables of the stack.

4.5 Case Study

In the following, we translate the concrete design of the stack in Chapter 3 into the refinement calculus.

4.5.1 States and Operations

As before, we abbreviate the state, input and output variables of the stack. These abbreviations are collected in Table 4.2.

Using the rules and strategies of the preceding sections, the state and operations are translated, and resultant abstract program is given in Figure 4.7. A possible design of the main program *MainProg* is given in the next section.

4.5.2 Main Program

For simplicity, we assume that the input stream of the system is a sequence of pairs of *COMMAND* and *OBJECT*. Each pair contains a request for push, pop or top, and an input object which is significant only for the push operation.

```
|[
    var $s : 1..max \rightarrow OBJECT$; $t : \mathbb{Z}$;
    and $0 \leq t \leq max$
    procedure $InitStackC \mathrel{\widehat{=}}$
        $s, t : [true\ ,\ t = 0]$

    procedure $PushC$(value $objI : OBJECT$; result $repO : REPORT$) $\mathrel{\widehat{=}}$
        if  $t < max \rightarrow$
                $s, t : [t < max\ ,\ t = t_0 + 1 \wedge s = s_0 \oplus \{t \mapsto objI\}]$;
                $repO : [true\ ,\ repO = ok]$
        []  $t = max \rightarrow repO : [t = max\ ,\ repO = full]$
        fi

    procedure $PopC$(result $repO : REPORT$) $\mathrel{\widehat{=}}$
        if  $t \neq 0 \rightarrow$
                $t : [t \neq 0\ ,\ t = t_0 - 1]$;
                $repO : [true\ ,\ repO = ok]$
        []  $t = 0 \rightarrow repO : [t = 0\ ,\ repO = empty]$
        fi

    procedure $TopC$(result $objO : OBJECT$; result $repO : REPORT$) $\mathrel{\widehat{=}}$
        if  $t \neq 0 \rightarrow$
                $objO : [t \neq 0\ ,\ objO = s(t)]$;
                $repO : [true\ ,\ repO = ok]$
        []  $t = 0 \rightarrow repO : [t = 0\ ,\ repO = empty]$
        fi

    •

    $InitStack$;
    $MainProgram$
]|
```

Figure 4.7: An abstract program translated from the concrete design of the stack.

$$COMMAND ::= push \mid pop \mid top$$

$$INPUT == \mathrm{seq}(COMMAND \times OBJECT)$$

Similarly, we assume that the output stream of the system is a sequence of pairs of *REPORT* and *OBJECT*. Each pair indicates the status of an operation and an output object which is significant only for the top operation.

$$OUTPUT == \mathrm{seq}(REPORT \times OBJECT)$$

We assume that the target programming language provides the following operators on sequences.

- *head*, which gives the first element of a sequence;
- *last*, which gives the last element of a sequence;
- *front*, which returns the sequence without its last element; and
- *tail*, which returns the sequence without its first element.

The programming language is also understood to have operators such as *first* and *second* which gives the first and second elements of an ordered pair.

$$\begin{array}{|l}
\hline
\textit{PushCommand} \\
\Delta StackC \\
\alpha, \alpha' : INPUT \\
\beta, \beta' : OUTPUT \\
\hline
first(head(\alpha)) = push \\
PushC[repO, objI \backslash first(last(\beta')), second(head(\alpha))] \\
\alpha' = tail\ \alpha \\
front\ \beta' = \beta \\
\hline
\end{array}$$

In *PushCommand*, the effect of a user request for pushing the stack is given. This is described in terms of the transformation of the input and output streams α and β. The effect on the stack is described by including *PushC* with the input and output variables appropriately renamed to associate with the input and output streams. The input stream is shortened by one command and output stream is lengthened with one output. The effects of popping and inquiring about the top of the stack are described in *PopCommand* and *TopCommand* respectively.

$$\begin{array}{|l}
\textit{PopCommand} \\
\hline
\Delta StackC \\
\alpha, \alpha' : INPUT \\
\beta, \beta' : OUTPUT \\
\hline
first(head(\alpha)) = pop \\
PopC[rep O \backslash first(last(\beta'))] \\
\alpha' = tail\ \alpha \\
front\ \beta' = \beta \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{TopCommand} \\
\hline
\Delta StackC \\
\alpha, \alpha' : INPUT \\
\beta, \beta' : OUTPUT \\
\hline
first(head(\alpha)) = top \\
TopC[rep O, obj O \backslash first(last(\beta')), second(last(\beta'))] \\
\alpha' = tail\ \alpha \\
front\ \beta' = \beta \\
\hline
\end{array}$$

Since each input must be a push, pop or top operation, the effect of consuming one input of the input sequence may be viewed as the disjunction of these three operations. This is described in *InputOutput*.

Operation	Precondition
$PushCommand$	$first(head(\alpha)) = push$
$PopCommand$	$first(head(\alpha)) = pop$
$TopCommand$	$first(head(\alpha)) = top$

Table 4.3: The preconditions of $PushCommand$, $PopCommand$, and $TopCommand$.

$$InputOutput \mathrel{\widehat{=}} PushCommand \vee PopCommand \vee TopCommand$$

Although this may not be immediately useful at this point, we give the translation of $InputOutput$. The preconditions of its three disjuncts may be found in Table 4.3. Using Rule "aiI", the specification statement

$$\alpha, \beta, t, s : [\text{pre } InputOutput \ , \ [\![InputOutput]\!]]$$

may be translated into the following.

```
if   $first(head(\alpha)) = push \rightarrow$
         $\alpha, \beta, s, t : [first(head(\alpha)) = push \ ,$
             $[\![PushC]\!][repO, objI \backslash first(last(\beta)), second(head(\alpha_0))] \wedge$
             $\alpha = tail \ \alpha_0 \wedge$
             $front \ \beta_0 = \beta]$;
[]   $first(head(\alpha)) = pop \rightarrow$
         $\alpha, \beta, s, t : [first(head(\alpha)) = pop \ ,$
             $[\![PopC]\!][repO \backslash first(last(\beta))] \wedge$
             $\alpha = tail \ \alpha_0 \wedge$
             $front \ \beta_0 = \beta]$;
[]   $first(head(\alpha)) = top \rightarrow$
         $\alpha, \beta, s, t : [first(head(\alpha)) = top \ ,$
             $[\![TopC]\!][repO, objO \backslash first(last(\beta)), second(last(\beta))]) \wedge$
             $\alpha = tail \ \alpha_0 \wedge$
             $front \ \beta_0 = \beta]$;
fi.
```

The main program of the system essentially applies the *InputOutput* operation until the input sequence is completely read. By capturing the operation as a relation, multiple application of an operation may be conveniently described using relational composition. Such a relation for *InputOutput* is given as *io* below.

$$STACKC == 1..max \rightarrow OBJECT$$

$$\begin{array}{|l}
io : STACKC \times \mathbb{Z} \times INPUT \times OUTPUT \\
\qquad \leftrightarrow STACKC \times \mathbb{Z} \times INPUT \times OUTPUT \\
\hline
io = \{ InputOutput \bullet (s, t, \alpha, \beta) \mapsto (s', t', \alpha', \beta') \}
\end{array}$$

The relation *io* may be understood as follows. If s, t, α, and β are the values of the current stack array, stack top, input and output streams, and s', t', α', and β' are the next stack array, stack top, input and output streams after executing *InputOutput* once, then the mapping

$$(s, t, \alpha, \beta) \mapsto (s', t', \alpha', \beta')$$

must be in the relation *io*.

We require that the *InputOutput* operation be performed for every command in the input stream. As such, we may relate initial and final states of the system by composing the relation *io* as many times as the length of the input sequence. This idea is captured in the schema *Main* which describes the execution of the system.

Main

$\Delta StackC$
$\alpha, \alpha' : INPUT$
$\beta, \beta' : OUTPUT$

$(s', t', \alpha', \beta') = io^{\#\alpha}(s, t, \alpha, \beta)$

Trivially, the translation of *Main* gives the specification statement

$$s, t, \alpha, \beta : [true \ , \ (s, t, \alpha, \beta) = io^{\#\alpha_0}(s_0, t_0, \alpha_0, \beta_0)].$$

In the next chapter, we show how the refinement of this statement may introduce the stack proceaures as well as the code translated from *InputOutput*.

4.6 Summary and Bibliographical Notes

In this chapter, we have examined many of the issues concerning the translation of a Z specification into the refinement calculus. The notation of the refinement calculus is introduced and the notion of algorithmic refinement within the framework of the calculus is summarized. A comparison of the two notations is then given while noting their relative suitability for specification and development work. Translation rules based on this comparison are then presented and more sophisticated derived rules for disjunction and conjunction of schemas are also given. We also give some directions on how to design a program that uses the procedures resulting from such a translation.

The basic techniques for translating from Z to the refinement calculus were proposed by King [25]. The version that we use is from Woodcock [45]. Some examples of translation may be found in [25, 45]

The notation of the refinement calculus that we use is from Morgan [31]. Other flavors of the refinement calculus may be found in [2, 35]. More references for the refinement calculus may be found in the last section of Chapter 5.

Chapter 5

Operation Refinement

Chapter 4 introduced the language of the refinement calculus and showed how the calculus may be used to develop programs. This chapter presents more refinement laws and gives examples to show how they may be used. As it is impossible to present all the laws that are available, a more complete list may be found in Appendix B.

5.1 Feasibility

An important concept in the refinement calculus is that of the feasibility of a specification, which indicates whether the specification may be refined to code. A specification is *feasible* if its precondition is at least as strong as the precondition that is calculated from that specification's postcondition (i.e., the *weakest* precondition.) This requires the precondition of a specification to have as least the constraints that are imposed by the postcondition, and this is stated formally in Definition "feas" below.

Definition 5.1 (feasibility "feas") The specification $w : [pre\ ,\ post]$ is *feasible* if and only if

$$(w = w_0) \land pre \land inv \ \Rightarrow\ (\exists\, w : T \bullet inv \land post),$$

where T is the type of w and inv is the invariant that is associated with the variables w during their declarations.

□

It is important to note that the calculus will not allow an infeasible specification to be refined into code. As such, it is impossible for an infeasible specification to lead to incorrect code, and hence, although possible to do so, it is not necessary for us to check the feasibility of specifications during development.

5.1.1 Pathological Specifications

In this section, we give some specifications which may be considered as extremes in the spectrum of specifications. Although these are not commonly used to describe programs (except for **skip**), they are very useful in understanding and explaining phenomena that may arise during a development.

abort

The specification statement

$$w : [\mathit{false}\ ,\ \mathit{true}]$$

is called **abort**. Since its precondition is false, it may not be used under any circumstance, and it is is never guaranteed to terminate. Even if it does terminate, the postcondition of *true* enables any result to be produced.

choose w

The specification statement

$$w : [\mathit{true}\ ,\ \mathit{true}]$$

is called **choose** w. Since its precondition is *true*, its invocation is always guaranteed to terminate, and since its postcondition is also *true*, it may produce any result.

skip

The specification statement

$$: [\mathit{true}\ ,\ \mathit{true}]$$

is called **skip**. This program is similar to **choose** w in that it is always guaranteed to terminate; however, it changes nothing as its frame is empty.

magic

The specification statement

$$w : [true\ ,\ false]$$

is called magic. Since its precondition is *true*, it is always guaranteed to terminate. However, since its postcondition is *false*, its terminating state can never be satisfied. As such, it establishes the impossible.

5.2 Some Basic Laws

In this section, we present some basic laws which enable the refinement of a specification into different language constructs.

5.2.1 Assignment

Our first law is one that introduces an assignment into the program.

Law 5.1 (assignment "ass") If $(w = w_0) \wedge pre \Rightarrow post[w \backslash E]$, then

$$w, x : [pre\ ,\ post] \quad \sqsubseteq \quad w := E.$$

□

Law "ass" states that a variable may be assigned a value if the replacement of the variable by that value in the postcondition represents a state that is derivable from its precondition.

Example 5.1 Since

$$\begin{array}{ll} & x = x_0 \wedge true \\ \Rightarrow & x + 1 > x_0 \\ \Leftrightarrow & x > x_0[x \backslash x + 1], \end{array}$$

$$\begin{array}{ll} & x : [true, x > x_0] \\ \sqsubseteq & \text{"ass"} \\ & x := x + 1. \end{array}$$

□

5.2.2 Local Block

Often during programming, we find the need to use some extra variables to hold intermediate values. The next law gives us a way to do this.

Law 5.2 (introduce local block "ilb") If w and x are disjoint, then

$$w : [pre\ ,\ post] \quad \sqsubseteq \quad |[\ \mathbf{var}\ x : T;\ \mathbf{and}\ inv \bullet w, x : [pre\ ,\ post]\ ]|.$$

□

Law "ilb" says that a fresh variable may be declared and included in the frame of a specification statement together with the introduction of a local block to contain its scope.

Example 5.2 Assume that we want to swap the values of two variables x and y of type T. We can introduce a variable t of the same type to hold one of their values when swapping.

$$
\begin{array}{ll}
 & x, y : [true \ , \ x = y_0 \wedge y = x_0] \\
\sqsubseteq & \text{"ilb"} \\
 & [\![\\
 & \quad \mathbf{var}\ t : T \bullet \\
 & \qquad x, y, t : [true \ , \ x = y_0 \wedge y = x_0] \\
 &]\!]
\end{array}
$$

□

5.2.3 Skip

If the precondition implies the postcondition, then a before-state that satisfies the precondition is also a legitimate after-state; as such, there is no need to do anything. This idea is contained in Law "sk" below.

Law 5.3 (skip command "sk") If $(w = w_0) \wedge pre \Rightarrow post$, then

$$w : [pre \ , \ post] \quad \sqsubseteq \quad \mathbf{skip}.$$

□

An avenue to understand this law is to convert the requirement $pre \Rightarrow post$ to $\neg pre \vee post$. Since the postcondition $post$ is guaranteed whenever the precondition pre is true, we are not obliged to do anything.

Existing laws may be used to derive new laws. This is particularly useful for building libraries of derived laws when a developer has established a preferred style of refinement either due to the target language or his mathematical intuitions. As an example, we show a derivation of Law "sk".

Example 5.3 A proof for Law "sk" is

$$
\begin{array}{ll}
 & w : [pre\ ,\ post] \\
\sqsubseteq & \text{"sp" and since } pre \Rightarrow post \\
 & w : [pre\ ,\ w = w_0] \\
\sqsubseteq & \text{"wp"} \\
 & w : [true\ ,\ w = w_0] \\
\sqsubseteq & \text{"efI"} \\
 & : [true\ ,\ true].
\end{array}
$$

Since skip is defined as $: [true\ ,\ true]$, our proof is complete.

□

5.2.4 Logical Constant

A *logical constant* may be introduced much like a variable, i.e., by declaring it within a local block. However, unlike a variable, the value taken by the constant is fixed, and since a logical constant is not an executable construct, it must be removed at the end of the development. Logical constants may be introduced to give names to some values that must exist. The value of a logical constant is often described in the precondition of a specification, where it may be understood that the constant takes on the value that makes the precondition true. Since logical constants are frequently used to hold the before-values of variables, an abbreviation has been formulated for this purpose.

Abbreviation 5.1 (initial variable "iv") Occurrences of 0-subscripted variables in the postcondition of a specification refer to values held by those variables in the *initial* state. Let x be any variable, probably occurring in the frame w. If X is a fresh name, and T is the type of x, then

$$\begin{array}{rl} & w : [pre\ ,\ post] \\ \widehat{=} & [\![\ \mathbf{con}\ X : T \bullet w : [pre \wedge x = X\ ,\ post[x_0 \backslash X]]\ ]\!]. \end{array}$$

We reserve 0-subscripted names for that purpose, and call them *initial variables.*

□

Example 5.4 Using Abbreviation "iv", the specification statement of Example 5.2 that swaps two variables x and y,

$$x, y, t : [true\ ,\ x = y_0 \wedge y = x_0],$$

may be written as

$$\begin{array}{l} [\![\\ \qquad \mathbf{con}\ X, Y \bullet \\ \qquad\qquad x, y, t : [x = X \wedge y = Y\ ,\ x = Y \wedge y = X] \\ ]\!] \end{array}$$

□

Logical constants may be removed at the end of a development by using Law "rlc" which is given below. This law is used to ensure the constant no longer appears in the program.

Law 5.4 (remove logical constant "rlc") If c occurs nowhere in program $prog$, then

$$|[\ \mathbf{con}\ c : T \bullet prog\ ]| \quad \sqsubseteq \quad prog.$$

□

5.2.5 Sequential Composition

A sequential composition may be introduced to divide a specification statement into two specification statements. This is accomplished by finding a single predicate to indicate the after-state of the first specification and the before-state of the second. By restricting the frame of the first specification to be a fraction of that of the original specification, the requirements of the original specification may be distributed between the two new specifications.

Law 5.5 (sequential composition "scII")

$$\begin{array}{ll} & w, x : [pre\ ,\ post] \\ \sqsubseteq & x : [pre\ ,\ mid]; \\ & w, x : [mid\ ,\ post]. \end{array}$$

The predicate mid must not contain initial variables, and $post$ must not contain x_0.

□

Example 5.5 We refine the specification of Example 5.4 to code. The strategy is to use the variable t to store the value of x during the swap of x and y.

$$\begin{array}{ll}
 & x, y, t : [x = X \land y = Y \ , \ x = Y \land y = X] \\
\sqsubseteq & \text{“scII”} \\
 & t : [x = X \land y = Y \ , \ x = X \land y = Y \land t = X]; \\
 & x, y, t : [x = X \land y = Y \land t = X \ , \ x = Y \land y = X]; \qquad \lhd \\
\sqsubseteq & \text{“scII”} \\
 & x : [x = X \land y = Y \land t = X \ , \ x = Y \land y = Y \land t = X]; \\
 & x, y, t : [x = Y \land y = Y \land t = X \ , \ x = Y \land y = X];
\end{array}$$

The symbol $\lhd$ is conventionally used to indicate the specification that is refined next. Collecting the leaves of the refinement tree, we have

$$\begin{array}{lll}
 & x, y : [x = X \land y = Y \ , \ x = Y \land y = X] & \\
\sqsubseteq & t : [x = X \land y = Y \ , \ x = X \land y = Y \land t = X]; & \text{(i)} \\
 & x : [x = X \land y = Y \land t = X \ , \ x = Y \land y = Y \land t = X]; & \text{(ii)} \\
 & x, y, t : [x = Y \land y = Y \land t = X \ , \ x = Y \land y = X]. & \text{(iii)}
\end{array}$$

Using Law “ass”, specifications (i), (ii) and (iii) may be easily refined to code.

$$\begin{array}{lll}
\text{(i)} & \sqsubseteq & t := x \\
\text{(ii)} & \sqsubseteq & x := y \\
\text{(iii)} & \sqsubseteq & y := t
\end{array}$$

□

5.2.6 Alternation

An alternation may be introduced by finding predicates which collectively cover the situations stated in the precondition. These predicates become the guards of the alternation, and since the precondition is assumed to be true when the alternation is executed, at least one of these guards will be true. Hence, we have a well-defined alternation which will not abort.

Law 5.6 (alternation "altI") If $pre \Rightarrow (\bigvee i \bullet G_i)$, then

$$\begin{array}{ll} & w : [pre\ ,\ post] \\ \sqsubseteq & \textbf{if } ([\!]\ i \bullet G_i \rightarrow w : [G_i \wedge pre\ ,\ post])\ \textbf{fi}. \end{array}$$

□

Example 5.6 The abstract program in Figure 4.3 that finds the maximum of two numbers may b. implemented with an alternation. Since $true \Rightarrow (x \geq y \vee y \geq x)$, we have

$$\begin{array}{l} x, y : [true\ ,\ (x_0 \geq y_0 \wedge x = y_0 \wedge y = x_0) \vee \\ \qquad\qquad\qquad (y_0 \geq x_0 \wedge x = x_0 \wedge y = y_0)] \end{array}$$

The symbol $\triangleq$ is used below to indicate an abbreviation where the postcondition of the starting specification was abbreviated as I.

$$\begin{array}{lll} \sqsubseteq & \text{"altI"} & \\ & I \triangleq (x_0 \geq y_0 \wedge x = y_0 \wedge y = x_0) \vee & \\ & \qquad\qquad (y_0 \geq x_0 \wedge x = x_0 \wedge y = y_0) \bullet & \\ & \textbf{if } x \geq y \rightarrow & \\ & \qquad x, y : [x \geq y\ ,\ I] & \triangleleft \\ & [\!]\ y \geq x \rightarrow & \\ & \qquad x, y : [y \geq x\ ,\ I] & \text{(i)} \\ & \textbf{fi} & \\ \sqsubseteq & \text{"sp" and then "wp"} & \\ & x, y : [true\ ,\ x = y_0 \wedge y = x_0] & \end{array}$$

$$\begin{array}{ll} \text{(i)} \sqsubseteq & \text{"sp" and then "wp"} \\ & x, y : [true\ ,\ x = x_0 \wedge y = y_0] \\ \sqsubseteq & \text{"sk"} \\ & \textbf{skip} \end{array}$$

Collecting the refinement leaves, we have

$$\begin{array}{ll} & x, y : [true\ ,\ (x_0 \geq y_0 \wedge x = y_0 \wedge y = x_0) \vee \\ & \qquad\qquad\qquad (y_0 \geq x_0 \wedge x = x_0 \wedge y = y_0)] \\ \sqsubseteq & \textbf{if } x \geq y \rightarrow \\ & \qquad x, y : [true\ ,\ x = y_0 \wedge y = x_0] \\ & [] \; y \geq x \rightarrow \\ & \qquad \textbf{skip} \\ & \textbf{fi}. \end{array}$$

□

5.2.7 Iteration

The central task of refining an iteration is to find an *invariant* which states what must be true during all repetitions. The refinement must also establish a *variant*, which is an expression that must decrease as the iteration progresses.

Law 5.7 (iteration "iter") Let *inv*, the *invariant*, be any predicate; let V, the *variant*, be any integer-valued expression. Then

$$\begin{array}{ll} & w : [inv\ ,\ inv \wedge \neg(\bigvee i \bullet G_i)] \\ \sqsubseteq & \textbf{do} \\ & \qquad ([] \; i \bullet G_i \rightarrow w : [inv \wedge G_i\ ,\ inv \wedge (0 \leq V < V_0)]) \\ & \textbf{od}. \end{array}$$

Note that neither *inv* nor G_i may contain initial variables and the expression V_0 is $V[w \backslash w_0]$.

□

The subtlety in this law lies with the formulation of the variant expression V. By requiring that V be non-negative and decreasing during each iteration, the user of the refinement is forced to consider the termination of the iteration. This consideration typically leads to the formulation of guards G_i which states exactly when the iteration may continue. These guards ensure that the iteration terminates before V becomes negative.

Example 5.7 We offer a refinement of the specification statement from Example 4.7 which checks the presence of an integer in an integer array. Our strategy is to check the elements of the array from the smallest index to the largest. If input is found, then the loop exits. Otherwise, the loop terminates after all of the elements are checked.

The specification statement of interest is

$$w : [true\ ,\ H]$$

where

$$H \triangleq (w = max + 1 \wedge x \notin array[1..max]) \vee (w \in 1..max \wedge array(w) = x).$$

This may be refined into an iteration which uses the variable w to hold the index of the array element that is currently being checked. Since the body of the iteration essentially increments w, and this is necessary only when input is not observed, we may formulate the invariant to say that the elements checked so far do not contain the input. This may be written as

$$I \triangleq x \notin array[1..w-1].$$

Since the variable w is increasing during each iteration and may be between 1 and $max + 1$, a variant expression may be

$$V \mathrel{\widehat{=}} max + 1 - w.$$

The exit condition is

$$\neg G \mathrel{\widehat{=}} w = max + 1 \vee array(w) = x$$

where G is the only guard of the iteration. These ideas are used in the following refinement.

```
    $w : [true\ ,\ H]$
$\sqsubseteq$  "scII"
    $I \mathrel{\widehat{=}} x \notin array[1..w-1] \bullet$
    $w : [true\ ,\ I];$                                        $\triangleleft$
    $w : [I\ ,\ H]$                                            (i)
$\sqsubseteq$  "ass"
    $w := 1$


(i) $\sqsubseteq$  "sp"
        $G \mathrel{\widehat{=}} w \neq max + 1 \wedge array(w) \neq x \bullet$
        $w : [I\ ,\ I \wedge \neg G]$
    $\sqsubseteq$  "iter" with invariant $I$ and variant $max + 1 - w$
        do $G \rightarrow$
            $w : [I \wedge G\ ,\ I \wedge (0 \leq max + 1 - w \leq max + 1 - w_0)]$   $\triangleleft$
        od
    $\sqsubseteq$  "ass"
        $w := w + 1$
```

□

5.2.8 Procedure

Parameterized procedures may be introduced through the mechanism of *substitution*. Three kinds of substitution are available: *call-by-value*, *call-by-result*, and *call-by-value-result*. The requirements for their use are given in the respective laws. We present here the last of the three. Since the law is quite unintuitive, a study of the example that follows may be necessary for a comprehension of the law.

Law 5.8 (value-result substitution "vrsII") If *post* does not contain a, then

$$\begin{array}{ll} & w \quad : [pre[f \backslash a]\ ,\ post[f_0, f \backslash a_0, a]] \\ \sqsubseteq & [\textbf{value result}\ f : T \backslash a] \bullet \\ & w, f : [pre\ ,\ post]. \end{array}$$

□

After a substitution law is applied, the formal parameters and the resulting specification statement may be combined to form a procedure. In their place, a procedure call with the actual parameters is introduced.

Example 5.8 Suppose that we have an abstract program that contains multiple specification statements of the kind

$$a, b : [true\ ,\ a = b_0 \wedge b = a_0]$$

which swaps the two variables a and b. It would be convenient to form a procedure that does this so that an occurrence of this specification may simply be replaced by a procedure call. In this way, instead of refining each occurrence of the specification, we are obligated to refine only that copy, which is the procedure. We show below how a procedure for the above specification and its call may be introduced into a program.

```
    a, b : [true , a = b_0 ∧ b = a_0]
=   a, b : [true , (x = y_0 ∧ y = x_0)][x_0, x, y_0, y\a_0, a, b_0, b]]
⊑   "vrsII"
    [value result x, y : ℤ\a, b] •
        x, y : [true , x = y_0 ∧ y = x_0]
=   procedure Swap(x, y : ℤ) ≙
        x, y : [true , x = y_0 ∧ y = x_0]                        ◁
    •
    Swap(a, b)
⊑   from the results of Example 5.2, Example 5.4, and
                Example 5.5 and using "rlc"
    |[
        var t : ℤ •
            t := x;
            x := y;
            y := t
    ]|
```

Collecting code, we have

```
procedure Swap(x, y : ℤ) ≙
    |[
        var t : ℤ •
            t := x;
            x := y;
            y := t
    ]|
•
Swap(a, b)
```

□

Duplication of Actual and Formal Parameters

Note that in all substitutions, if f is a list of formal parameters then it must not contain repeated variables, because a substitution of the kind $[y, y \backslash 1, 2]$ would be meaningless. For the same reason, since $[a \backslash f]$ occurs in value-result and result substitutions, the actual parameters a must not contain repeated variables.

Variable Capture

It is often desirable to group all the procedures together in the outermost block of the complete program. This may be necessary due to the requirements of the target programming language. One possible difficulty with moving a procedure is that it might move variables into and out of the blocks in which they are declared. As such, it is recommended that a procedure use only variables that are either global, i.e., whose scope extend throughout the whole program, or local within the body of the procedure.

Substitution by Reference

The most common substitution techniques used in current programming languages are *call-by-value* and *call-by-reference*. Call-by-reference substitution may be effectively modeled by value-result substitution except when there is *aliasing*, i.e., when two distinct names in the procedure are used to refer to one single variable [31, 29].

Aliasing in call-by-reference occurs explicitly in

$$[\textbf{reference } x, y \backslash z, z]$$

where x and y are both used to refer to z. With call-by-reference, a change of y in the procedure changes x and z as well. On the other hand, in a similar call-by-value-result substitution, a change of y in the procedure does not affect x, and upon the exit of the procedure, z will be assigned the value of either x or y. An example of implicit aliasing is

$$x := y^2 \; [\textbf{reference } y \backslash x].$$

An execution of this with call-by-reference will enable x to square itself, while a similar call-by-value-result substitution will prevent the value of x from changing.

By avoiding occurrence of aliasing, we may use call-by-value-result to develop programs that contain call-by-reference substitutions. The explicit case of aliasing may be avoided by disallowing repeated variables in the parameter list of any value-result substitution. Note that from the discussion of a previous section on the duplication of actual parameters, we have already disallowed duplication of variables in actual parameter list for value-result substitutions. The implicit case of aliasing may be dealt with by simply requiring that an actual parameter does not appear in the code of the procedure.

5.3 Case Study

In the following, we give one refinement of the procedures and main program of the stack example of Chapter 4.

5.3.1 Procedures

Since the refinement of the procedures is easy, we show here only the process for procedure *PushC*. All resultant code for the program, except that for the main program, is collected in Figure 5.1.

<u>**Refinement of the procedure *PushC***</u>

The specification statements in the procedure *PushC* are refined below. First, we refine the first specification statement in the first branch of the alternation of *PushC* from Figure 4.7.

$$
\begin{array}{ll}
 & s, t : \left[t < max \, , \, \begin{array}{c} t = t_0 + 1 \wedge \\ s = s_0 \oplus \{ t \mapsto objI \} \end{array} \right] \\
\sqsubseteq & \text{“scI”} \\
 & \textbf{con } T \bullet \\
 & t : [t < max - 1 \, , \, t = t_0 + 1]; \qquad\qquad \lhd \\
 & s, t : \left[t = T + 1 \, , \, \begin{array}{c} t = T + 1 \wedge \\ s = s_0 \oplus \{ t \mapsto objI \} \end{array} \right] \qquad (\text{i}) \\
\sqsubseteq & \text{“ass”} \\
 & t := t + 1
\end{array}
$$

```
var $s : 1..max \to OBJECT$; $t : \mathbb{Z}$;
and $0 \le t \le max$
procedure $InitStackC \triangleq t := 0$
procedure $PushC$(value $objI : OBJECT$; result $repO : REPORT$) $\triangleq$
    if  $t < max \to$
            $t := t + 1$;
            $s(t) := objI$;
            $repO := ok$
    []  $t = max \to$
            $repO := full$
    fi;
procedure $PopC$(result $repO : REPORT$) $\triangleq$
    if  $t \neq 0 \to$
            $t := t - 1$;
            $repO := ok$
    []  $t = 0 \to$
            $repO := empty$
    fi;
procedure $TopC$(result $objO : OBJECT$; result $repO : REPORT$) $\triangleq$
    if  $t \neq 0 \to$
            $objO := s(t)$
            $repO := ok$
    []  $t = 0 \to$
            $repO := empty$
    fi
•
$InitStackC$;
$MainProgram$
```

Figure 5.1: An abstract program of the stack with refined procedures.

$$
\begin{array}{ll}
(i) = & \text{"sp" both ways} \\
& s,t : \left[t = T+1 \ , \ \begin{array}{c} t = T+1 \wedge \\ \{t\} \lhd s = \{t\} \lhd s_0 \wedge \\ s(t) = objl \end{array} \right] \\
\sqsubseteq & \text{"ass"} \\
& s(t) := objl
\end{array}
$$

The refinement of the second specification statement of the first branch is given next.

$$
\begin{array}{ll}
& repO : [true \ , \ repO = ok] \\
\sqsubseteq & \text{"ass"} \\
& repO := ok
\end{array}
$$

Finally, we refine the specification in the second branch of the alternation.

$$
\begin{array}{ll}
& repO : [t = max - 1 \ , \ repO = full] \\
\sqsubseteq & \text{"ass"} \\
& repO := full
\end{array}
$$

5.3.2 Main Program

We describe below a possible refinement of the main program. Recall that this program has the specification

$$
s, t, \alpha, \beta : [true \ , \ (s, t, \alpha, \beta) = io^{\#\alpha_0}(s_0, t_0, \alpha_0, \beta_0)].
$$

Using the abbreviation for initial variable, we rewrite this specification as

$=$ "iv"
con $S, T, A, B \bullet$

$$s, t, \alpha, \beta : \left[\begin{array}{l} S = s \wedge \\ T = t \wedge \\ A = \alpha \wedge \\ B = \beta \end{array} , \ (s, t, \alpha, \beta) = io^{\# A}(S, T, A, B) \right]$$

We want our program to continuously read a command - input object pair, and execute the relevant operation, until no more input is found. Clearly, this involves an iteration with a terminating condition indicating that the input stream is empty, and a variant expression that gives the length of the input stream. The next few steps are the typical ones for setting up such an iteration.

$\sqsubseteq$ "wp"
$s, t, \alpha, \beta : [(s, t, \alpha, \beta) = io^{\# A - \# \alpha}(S, T, A, B) \ ,$
$\qquad (s, t, \alpha, \beta) = io^{\# A}(S, T, A, B)]$

$\sqsubseteq$ "sp"
$I \mathrel{\hat=} (s, t, \alpha, \beta) = io^{\# A - \# \alpha}(S, T, A, B) \bullet$
$s, t, \alpha, \beta : [I \ , \ I \wedge \alpha = \langle\rangle]$

$\sqsubseteq$ "isg" with invariance I and variance $\# \alpha$
```
do $\alpha \neq \langle\rangle \rightarrow$
    $s, t, \alpha, \beta : [\alpha \neq \langle\rangle \wedge I \ , \ I \wedge 0 \leq \# \alpha \leq \# \alpha_0]$    $\triangleleft$
od
```

The specification in the body of the iteration may be refined to introduce the abstract program for operation *InputOutput*.

$\sqsubseteq$ "sp"
$s, t, \alpha, \beta : [\alpha \neq \langle\rangle \wedge I \ , \ I \wedge \# \alpha = \# \alpha_0 - 1]$

```
if  $first(head(\alpha)) = push \rightarrow$
        $\alpha, \beta, s, t : [first(head(\alpha)) = push\ ,$
                $[\![PushC]\!][repO, objI \backslash first(last(\beta)), second(head(\alpha_0))] \wedge$
                $\alpha = tail\ \alpha_0 \wedge$
                $front\ \beta = \beta_0];$
[]  $first(head(\alpha)) = pop \rightarrow$
        $\alpha, \beta, s, t : [first(head(\alpha)) = pop\ ,$
                $[\![PopC]\!][repO \backslash first(last(\beta))] \wedge$
                $\alpha = tail\ \alpha_0 \wedge$
                $front\ \beta = \beta_0];$
[]  $first(head(\alpha)) = top \rightarrow$
        $\alpha, \beta, s, t : [first(head(\alpha)) = top\ ,$
                $[\![TopC]\!][repO, objO \backslash first(last(\beta)), second(last(\beta))]) \wedge$
                $\alpha = tail\ \alpha_0 \wedge$
                $front\ \beta = \beta_0];$
fi.
```

Figure 5.2: An abstract program translated from the schema *InputOutput*.

$\sqsubseteq$ "sp" and then "wp"

$$s, t, \alpha, \beta : \left[\alpha \neq \langle\rangle\ ,\ \begin{array}{c} (s, t, \alpha, \beta) = io(s_0, t_0, \alpha_0, \beta_0) \wedge \\ \#\alpha = \#\alpha_0 - 1 \end{array} \right]$$

$\sqsubseteq$ "sp"

$$s, t, \alpha, \beta : \left[\alpha \neq \langle\rangle\ ,\ \begin{array}{c} [\![InputOutput]\!] \wedge \\ \#\alpha = \#\alpha_0 - 1 \end{array} \right]$$

$\sqsubseteq$ "sp"

$$s, t, \alpha, \beta : [\alpha \neq \langle\rangle\ ,\ [\![InputOutput]\!]]$$

Using the refinement in Section 4.5.2 for *InputOutput*, we can refine the above into the program in Figure 5.2, which gives the body of the iteration.

A Refinement to Introduce *PopC*

The abstract program in Figure 5.2 may be refined to introduce the procedures of the stack. We show here how to refine the second branch of the alternation to introduce procedure *PopC*. The other branches may be refined similarly. The specification in the second branch of the alternation is

$$\alpha, \beta, s, t : \left[first(head(\alpha)) = pop \;, \quad \begin{array}{c} [\![PopC]\!][repO \backslash first(last(\beta))] \wedge \\ \alpha = tail\ \alpha_0 \wedge \\ front\ \beta = \beta_0 \end{array} \right]$$

We introduce a variable to hold the output of the *PopC* operation, and decompose this specification into a specification that performs the pop operation and another that interacts with the input and output streams.

$$\begin{array}{ll} \sqsubseteq & \text{"ilb", "sp" and then "wp"} \\ & |[\ \mathbf{var}\ r : REPORT;\ obj : OBJECT \bullet \\ & \qquad \alpha, \beta, s, t, r, obj : \left[true \;, \quad \begin{array}{c} [\![PopC]\!][repO \backslash r] \wedge \\ (r, obj) = last(\beta) \wedge \\ \alpha = tail\ \alpha_0 \wedge \\ front\ \beta = \beta_0 \end{array} \right] \qquad \lhd \\ &]| \end{array}$$

$$\begin{array}{ll} \sqsubseteq & \text{"scI"} \\ & \mathbf{con}\ S, T, R, OBJ \bullet \\ & \qquad s, t, r, obj : [true \;,\ [\![PopC]\!][repO \backslash r]; \qquad\qquad (i) \\ & \qquad \alpha, \beta, s, t, r, obj : [[\![PopC]\!][repO \backslash r][s, t, r, obj \backslash S, T, R, OBJ] \;, \\ & \qquad\qquad \left. \begin{array}{c} [\![PopC]\!][repO \backslash r][s_0, t_0, r_0, obj_0 \backslash S, T, R, OBJ] \wedge \\ (r, obj) = last(\beta) \wedge \\ \alpha = tail\ \alpha_0 \wedge \\ front\ \beta = \beta_0 \end{array} \right] \qquad \lhd \end{array}$$

$$\begin{array}{ll} \sqsubseteq & \alpha := tail\ \alpha; \\ & \beta := \beta \frown \langle (r, obj) \rangle. \end{array}$$

Specification (i) may be refined to introduce the procedure $PopC$ by applying Law "rs".

$$
\begin{array}{lll}
(i) & \sqsubseteq & \text{"rs"} \\
 & & s, t, obj, rcpO : [true\ ,\ [\![PopC]\!]][\textbf{result}\ rcpO \backslash r]
\end{array}
$$

$$
\begin{array}{ll}
= & \textbf{procedure}\ PopC(\textbf{result}\ rcpO : REPORT) \mathrel{\widehat{=}} \\
 & \quad s, t, rcpO : [true, [\![PopC]\!]] \\
 & \bullet \\
 & PopC(r)
\end{array}
$$

Since procedure $PopC$ uses only variables that are either global or local to $PopC$, the procedure may be moved to the outermost block. For completeness, the code for our stack program is given in Figure 5.3.

5.4 Summary and Bibliographical Notes

This chapter contains several basic laws of the refinement calculus and examples to show their use. These laws allow many of the major executable constructs to be introduced during the refinement of a specification.

The material presented in this chapter may be found in Morgan's book on the refinement calculus [31]. In this book, Morgan also treats refinement into modules, recursion, and data refinement within the framework of the refinement calculus. Theoretical discussions on the different aspects of the calculus may be found in [33, 30] (specification statement), [29] (procedures and parameters), [32] (types and invariants), and [34, 28, 27] (data refinement).

```
var $s : 1..max \rightarrow OBJECT$;
    $t : \mathbb{Z}$;
    $r : REPORT$;
    $obj : OBJECT$
and $0 \leq t \leq max$
procedure $InitStackC \triangleq t := 0$
procedure $PushC$(value $objI : OBJECT$; result $repO : REPORT$) $\triangleq$
    if   $t < max \rightarrow$
             $t := t + 1$;
             $s(t) := objI$;
             $repO := ok$
    []   $t = max \rightarrow$
             $repO := full$
    fi;
procedure $PopC$(result $repO : REPORT$) $\triangleq$
    if   $t \neq 0 \rightarrow$
             $t := t - 1$;
             $repO := ok$
    []   $t = 0 \rightarrow$
             $repO := empty$
    fi;
procedure $TopC$(result $objO : OBJECT$; result $repO : REPORT$) $\triangleq$
    if   $t \neq 0 \rightarrow$
             $objO := s(t)$
             $repO := ok$
    []   $t = 0 \rightarrow$
             $repO := empty$
    fi
•
$InitStackC$;
do $\alpha \neq \langle\rangle \rightarrow$
    if  $first(head(\alpha)) = push \rightarrow PushC(second(head(\alpha)), r)$
    []  $first(head(\alpha)) = pop \rightarrow PopC(r)$
    []  $first(head(\alpha)) = top \rightarrow TopC(obj, r)$
    fi;
    $\alpha := tail\ \alpha$;
    $\beta := \beta \frown \langle(r, obj)\rangle$
od
```

Figure 5.3: Code calculated from the abstract program of the stack.

One of the difficulties associated with the use of the refinement calculus is the derivation of loop invariants (see Law "iter"). Some discussion on how the obtain loop invariants may be found in [13].

Wordsworth has suggested an approach to operation refinement that avoids the refinement calculus [47]. Wordsworth's method which also enable code in guarded commands to be yielded from a concrete design involves stating an algorithm design and proving its correctness. The state-and-prove nature of his approach complements the calculative nature of the refinement calculus.

Chapter 6

Case Study: The Paragraph Problem

This chapter contains a non-trivial case study. Besides showing how formal methods may be appropriately used to manage the algorithmic complexity in the development of software systems, this case study also indicates some directions on how predefined programming language and library routines may be introduced into our framework of formal development.

6.1 Even Paragraphs

The problem for this case study is that of laying out words into lines such that these lines form an *even paragraph*. To explain what an even paragraph is, we borrow some examples from Morgan [31, pages 170–171]. In a *simple* paragraph

```
|Compare the paragraphs of Figure 6.1 and      |
|Figure 6.2. In simple paragraphs, like Figure |
|6.1, each line is filled as much as possible  |
|before moving on to the next. As a            |
|consequence, the minimum number of lines is   |
|used; but a long word arriving near the end of|
|a line can cause a large gap there.           |
```

Figure 6.1: A simple paragraph.

```
|Compare the paragraphs of Figure 6.1 and      |
|Figure 6.2. In simple paragraphs, like        |
|Figure 6.1, each line is filled as much       |
|as possible before moving on to the next.     |
|As a consequence, the minimum number of       |
|lines is used; but a long word arriving       |
|near the end of a line can cause a large      |
|gap there.                                    |
```

Figure 6.2: An even paragraph.

(see Figure 6.1), each line is filled with as many words as possible before the next line is filled. Although this scheme minimizes the number of lines used, it may require some lines to end with a large number of white spaces. This happens when the next word of a line is long and cannot be fitted as the last word of that line. An *even* paragraph (see Figure 6.2) differs from a simple one in that the number of white spaces of a short line is reduced by distributing some of these spaces over earlier longer lines.

This problem was stated by Bird [5], and was specified and partially refined by Morgan using the refinement calculus [31]. In the following, we show how a program in the programming language Pascal [10] that computes even paragraphs may be derived using the formal software development process that is advocated in this thesis. For the sake of brevity, we omit many of the proof and derivation details, and only mention important strategies.

6.2 Abstract Specification

The global constants *maxWord* and *maxLength* are used to denote the maximum number of words and the maximum length of each line in a paragraph.

$$\begin{array}{|l} maxWord : \mathbb{N} \end{array}$$

$$\begin{array}{|l} maxLength : \mathbb{N} \\ \hline maxLength \geq 1 \end{array}$$

$[CHAR]$

$$\begin{array}{|l} \mathsf{newline}, \mathsf{tab}, \mathsf{space} : CHAR \\ \hline \mathsf{newline} \neq \mathsf{tab} \\ \mathsf{tab} \neq \mathsf{space} \\ \mathsf{newline} \neq \mathsf{space} \end{array}$$

The set *CHAR* is declared to represent the set of characters allowable in a paragraph. Using this, we define a *word* as a non-empty sequence of at most *maxLength* characters, which does not contain any newline, tab or space characters. These words are contained in the set *WORD*. For convenience, we will refer to newline, tab, and space characters as *white spaces*.

$$WORD == \{\ w : \mathrm{seq}\ CHAR \mid 0 < \#w \leq maxLength \land \\ \mathrm{ran}\ w \cap \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} = \emptyset\}$$

6.2.1 State Space and Initial States

The state space and initial states of the the system are described in EP and $InitEP$[1]. The system maintains a sequence of at most $maxWord$ words which is initially empty.

$$\begin{array}{|l}
\hline
EP \\
words : \mathrm{seq}\ WORD \\
\hline
\#words \leq maxWord \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
InitEP \\
EP' \\
\hline
words' = \langle\rangle \\
\hline
\end{array}$$

6.2.2 Operations

For simplicity, we may regard the input to and output from the system as sequences of characters.

$$INPUT == \mathrm{seq}\ CHAR \\ OUTPUT == \mathrm{seq}\ CHAR$$

[1] Traditionally, the paragraph problem has been specified in terms of a relation between the input and output sequences. We adopt a state space specification so as to illustrate our method of software development.

Using *INPUT* and *OUTPUT*, the operations for reading words from an input and writing an even paragraph onto an output is described below.

Read Words

Functions *conS* and *conW* remove leading white spaces and non white spaces from an input, respectively. Function *retW*, which is similarly formulated, returns the longest sequence of leading non white-space characters.

$$
\begin{array}{|l}
conS : INPUT \rightarrow INPUT \\
\hline
\forall s : INPUT \bullet \\
\quad (s = \langle\rangle \lor head\ s \notin \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \Rightarrow \\
\qquad\qquad\qquad\qquad conS(s) = s) \\
\quad \land \\
\quad (s \neq \langle\rangle \land head\ s \in \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \Rightarrow \\
\qquad\qquad\qquad\qquad conS(s) = conS(tail\ s))
\end{array}
$$

$$
\begin{array}{|l}
conW : INPUT \rightarrow INPUT \\
\hline
\forall s : INPUT \bullet \\
\quad (s = \langle\rangle \lor head\ s \in \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \Rightarrow \\
\qquad\qquad\qquad\qquad conW(s) = s) \\
\quad \land \\
\quad (s \neq \langle\rangle \land head\ s \notin \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \Rightarrow \\
\qquad\qquad\qquad\qquad conW(s) = conW(tail\ s))
\end{array}
$$

$$
\begin{array}{|l}
retW : INPUT \rightarrow \mathrm{seq}\ CHAR \\
\hline
\forall s : INPUT \bullet \\
\quad (s = \langle\rangle \lor head\ s \in \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \Rightarrow \\
\qquad\qquad\qquad\qquad retW(s) = \langle\rangle) \\
\quad \land \\
\quad (s \neq \langle\rangle \land head\ s \notin \{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \Rightarrow \\
\qquad\qquad\qquad\qquad retW(s) = \langle head\ s\rangle \frown retW(tail\ s))
\end{array}
$$

With the assumption that words are separated by at least one white space, a function called $formWS$ is defined which extracts words from an input and returns a sequence of type $WORD$ that contains those words. As shown in its definition, the function $formWS$ uses the functions $conS$, $conW$, and $retW$.

$$
\begin{array}{|l}
formWS : INPUT \rightarrow \text{seq } WORD \\
\hline
\forall s : INPUT \bullet \\
\quad (conS(s) = \langle\rangle \Rightarrow \\
\qquad formWS(s) = \langle\rangle) \\
\quad \wedge \\
\quad (conS(s) \neq \langle\rangle \Rightarrow \\
\qquad formWS(s) = \langle(1..maxLength) \lhd retW(conS(s))\rangle ^\frown \\
\qquad\qquad formWS(conW(conS(s))))
\end{array}
$$

Note that when a word is returned by function $retW$, $formWS$ truncates it if that word is longer than $maxLength$. Thus, a word that is accepted by $formWS$ is always of type $WORD$. Using function $formWS$, the operation of reading an input is merely an application of $formWS$ on the input. The word sequence that is yielded from reading the input is also truncated to ensure that the system stores only the first $maxWord$ words.

$$
\begin{array}{|l}
\hline
ReadInput \\
input? : \text{seq } CHAR \\
\Delta EP \\
\hline
words' = (1..maxWord) \lhd formWS(input?) \\
\hline
\end{array}
$$

Lines and Paragraphs

Given a sequence of words, the function *width* computes the length of a line that is made up of these words with a space separating each pair of consecutive words.

$$\begin{array}{|l}
width : \mathrm{seq}\ WORD \rightarrow \mathbb{N} \\
\hline
\forall\, ws : \mathrm{seq}\ WORD \bullet \\
\quad (ws = \langle\rangle \Rightarrow \\
\qquad width(ws) = 0) \wedge \\
\quad (ws \neq \langle\rangle \Rightarrow \\
\qquad width(ws) = (\#ws - 1) + \sum_{k=1}^{\#ws} \#(ws(k)))
\end{array}$$

Using function *width*, we define a line to be a sequence of words with a width of at most *maxLength*.

$$LINE == \{\ l : \mathrm{seq}\ WORD \mid 1 \leq width(l) \leq maxLength\ \}$$

Subsequently, a paragraph is easily defined as a sequence of lines.

$$PARAGRAPH == \mathrm{seq}\ LINE$$

Waste and Even Paragraphs

The *waste* of a paragraph is the maximum number of rightmost white spaces that are contained in any line of the paragraph, except the last. Function *waste* computes the waste of a paragraph.

$$\begin{array}{|l}
waste : PARAGRAPH \rightarrow \mathbb{N} \\
\hline
\forall p : PARAGRAPH \bullet \\
\quad (\#p \leq 1 \Rightarrow \\
\qquad waste(p) = 0) \land \\
\quad (\#p > 1 \Rightarrow \\
\qquad waste(p) = max \; \{ \; l : LINE \mid l \in \mathrm{ran}(front \; p) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad maxLength - width(l) \; \})
\end{array}$$

The minimum waste of a sequence of words is the minimum waste of a paragraph that contains these words. Minimum waste is computed by function *minWaste*.

$$\begin{array}{|l}
minWaste : \mathrm{seq}\ WORD \rightarrow \mathbb{N} \\
\hline
\forall ws : \mathrm{seq}\ WORD \bullet \\
\quad minWaste(ws) = min \; \{p : PARAGRAPH \mid {}^\frown/\, p = ws \bullet waste(p)\}
\end{array}$$

The relation *evenP* relates a sequence of words and a paragraph, where the paragraph is a layout of these words, and has a waste that is equal to the minimum waste of the sequence.

$$\begin{array}{|l}
\_\ evenP\ \_ : \mathrm{seq}\ WORD \leftrightarrow PARAGRAPH \\
\hline
\forall ws : \mathrm{seq}\ WORD;\ p : PARAGRAPH \bullet \\
\quad ws \ evenP \ p \;\; \Leftrightarrow \;\; {}^\frown/\, p = ws \land waste(p) = minWaste(ws)
\end{array}$$

Computing and Writing Even Paragraphs

Functions *insertS* and *formOutput* indicate how a paragraph should be laid out. These functions ensure that each consecutive pair of words in a line are separated by one space, and that each line including the last ends with a newline character.

$$
\begin{array}{|l}
insertS : LINE \rightarrow \mathrm{seq}\ CHAR \\
\hline
\forall l : LINE \bullet \\
\quad (\#l = 1 \Rightarrow \\
\qquad\qquad insertS(l) = last\ l) \\
\quad \wedge \\
\quad (\#l > 1 \Rightarrow \\
\qquad\qquad insertS(l) = (head\ l) \frown \langle \mathsf{space} \rangle \frown insertS(tail\ l))
\end{array}
$$

$$
\begin{array}{|l}
formOutput : PARAGRAPH \rightarrow OUTPUT \\
\hline
\forall p : PARAGRAPH \bullet \\
\quad (\#p = 0 \Rightarrow \\
\qquad\qquad formOutput(p) = \langle\rangle) \\
\quad \wedge \\
\quad (\#p \geq 1 \Rightarrow \\
\qquad\qquad formOutput(p) = insertS(head\ p) \frown \\
\qquad\qquad\qquad\qquad \langle \mathsf{newline} \rangle \frown formOutput(tail\ p))
\end{array}
$$

Using the preceding function definitions, the operation *WriteParagraph* may now be easily described as outputting a paragraph that is an even layout of the words stored in the system.

$$
\begin{array}{|l}
\hline
WriteParagraph \\
\Xi EP \\
output! : OUTPUT \\
\hline
\exists p : PARAGRAPH \mid \\
\quad words\ evenP\ p \bullet \\
\qquad output! = formOutput(p) \\
\hline
\end{array}
$$

6.3 Concrete Design

We propose a concrete design that uses data structures that are available in Pascal. We find it convenient to define a word as a record with an array of characters and an integer to store the word and its length, respectively. This is modeled in schema $WordC$.

$$CHARARRAY == 1..maxLength \rightarrow CHAR$$

$$\begin{array}{|l}
\textit{WordC} \\
\hline
word : CHARARRAY \\
length : \mathbb{Z} \\
\hline
0 \leq length \leq maxLength \\
\{\mathsf{newline}, \mathsf{tab}, \mathsf{space}\} \cap \mathrm{ran}(1..length \lhd word) = \varnothing \\
\hline
\end{array}$$

The use of a schema as a type allows $WordC$ to be viewed as the set of tuples of *word* and *length* that satisfy the predicate in $WordC$. Using *schema projection*, the components of a schema object may be referenced in a similar manner as the fields of a Pascal record. For instance, if w is declared as having type $WordC$, then $w.word$ will allow us to refer to the word component of w.

The system state space EPC may be modeled as an array of $WordC$ with an integer variable $totalC$ to indicate the number of words present in the system.

$$\begin{array}{|l}
\textit{EPC} \\
\hline
wordsC : 1..maxWord \rightarrow WordC \\
totalC : \mathbb{Z} \\
\hline
0 \leq totalC \leq maxWord \\
\hline
\end{array}$$

Clearly, the system when started should contain no words.

$$\begin{array}{|l}
\textit{InitEPC} \\
\hline
EPC' \\
\hline
totalC' = 0 \\
\hline
\end{array}$$

Read Words

The strategy for reading words from an input in this concrete design is the same as that in the abstract specification[2].

$$
\begin{array}{|l}
conSC : INPUT \rightarrow INPUT \\
\hline
conSC = conS
\end{array}
$$

$$
\begin{array}{|l}
conWC : INPUT \rightarrow INPUT \\
\hline
conWC = conW
\end{array}
$$

$$
\begin{array}{|l}
retWC : INPUT \rightarrow \mathrm{seq}\ CHAR \\
\hline
retWC = retW
\end{array}
$$

However, the way to store these words in the system is quite different.

$$
\begin{array}{|l}
\hline
ReadInputC \\
input? : INPUT \\
\Delta EPC \\
\hline
totalC' = \\
\quad min\ \{ \\
\quad\quad min\ \{\ n : \mathbb{N} \mid conSC(conWC \circ conSC)^{n}(input?) = \langle\rangle\ \}, \\
\quad\quad maxWord \\
\quad\quad \} \\
\forall i : 1..totalC' \bullet \\
\quad wordsC'(i).length = \\
\quad\quad \#(1..maxLength \lhd \\
\quad\quad\quad retWC(conSC((conWC \circ conSC)^{i-1}(input?)))) \land \\
\quad wordsC'(i).word = wordsC(i).word \oplus \\
\quad\quad (1..maxLength \lhd \\
\quad\quad\quad retWC(conSC((conWC \circ conSC)^{i-1}(input?)))) \\
\hline
\end{array}
$$

[2]The functions *conSC*, *conWC*, and *retWC* are redundant. They are presented to satisfy our naming conventions.

Lines and Paragraphs

Lines and paragraphs in our concrete design are defined similar to those in the abstract specification.

$$
\begin{array}{|l}
widthC : \mathrm{seq}\ WordC \rightarrow \mathbb{N} \\
\hline
\forall\, wCs : \mathrm{seq}\ WordC\ \bullet \\
\quad (wCs = \langle\rangle \Rightarrow \\
\qquad widthC(wCs) = 0) \\
\quad \wedge \\
\quad (wCs \neq \langle\rangle \Rightarrow \\
\qquad widthC(wCs) = (\#wCs - 1) + \sum_{k=1}^{\#wCs} wCs(k).length)
\end{array}
$$

$$LINEC == \{\ lC : \mathrm{seq}\ WordC \mid 1 \leq widthC(lC) \leq maxLength\ \}$$

$$PARAGRAPHC == \mathrm{seq}\ LINEC$$

Waste and Even Paragraphs

The concrete version of waste, minimum waste, and even paragraphs are defined similar to their abstract version.

$$
\begin{array}{|l}
wasteC : PARAGRAPHC \rightarrow \mathbb{N} \\
\hline
\forall\, pC : PARAGRAPHC\ \bullet \\
\quad (\#pC \leq 1 \Rightarrow \\
\qquad wasteC(pC) = 0) \\
\quad \wedge \\
\quad (\#pC > 1 \Rightarrow \\
\qquad wasteC(pC) = max\ \{lC : LINEC \mid \\
\qquad\qquad\qquad\qquad lC \in \mathrm{ran}(front\ pC)\ \bullet \\
\qquad\qquad\qquad\qquad\quad maxLength - widthC(lC)\})
\end{array}
$$

$$\begin{array}{|l}
minWasteC : \mathrm{seq}\ WordC \rightarrow \mathbb{N} \\
\hline
\forall wCs : \mathrm{seq}\ WordC \bullet \\
\quad minWasteC(wCs) = \\
\quad min\ \{pC : PARAGRAPHC \mid \frown/\ pC = wCs \bullet wasteC(pC)\}
\end{array}$$

$$\begin{array}{|l}
\_\ evenPC\ \_ : \mathrm{seq}\ WordC \leftrightarrow PARAGRAPHC \\
\hline
\forall wCs : \mathrm{seq}\ WordC;\ pC : PARAGRAPHC \bullet \\
\quad wCs\ evenPC\ pC \quad \Leftrightarrow \\
\qquad\qquad \frown/\ pC = wCs \land wasteC(pC) = minWasteC(wCs))
\end{array}$$

Writing Even Paragraphs

The only difference in the specification of outputting an even paragraph is the addition of a function *getWordC* to extract the word that is contained in an item of type *WordC*.

$$\begin{array}{|l}
getWordC : WordC \rightarrow WORD \\
\hline
\forall wC : WordC \bullet \\
\quad getWordC(wC) = 1..wC.length \lhd wC.word
\end{array}$$

$$\begin{array}{|l}
insertSC : LINEC \rightarrow \mathrm{seq}\ CHAR \\
\hline
\forall lC : LINEC \bullet \\
\quad \#lC = 1 \Rightarrow \\
\qquad\qquad insertSC(lC) = getWordC(last\ lC) \\
\land \\
\quad \#lC > 1 \Rightarrow \\
\qquad\qquad insertSC(lC) = \\
\qquad\qquad\quad getWordC(head\ lC) \frown \langle \mathsf{space} \rangle \frown insertSC(tail\ lC)
\end{array}$$

$$\begin{array}{l}
formOutputC : PARAGRAPHC \rightarrow OUTPUT \\
\hline
\forall pC : PARAGRAPHC \bullet \\
\quad (\#pC = 0 \Rightarrow \\
\qquad\qquad formOutputC(pC) = \langle\rangle) \\
\quad \wedge \\
\quad (\#pC \geq 1 \Rightarrow \\
\qquad formOutputC(pC) = insertSC(head\ pC)^\frown \\
\qquad\qquad\qquad \langle \mathsf{newline} \rangle ^\frown formOutputC(tail\ pC))
\end{array}$$

$$\begin{array}{l}
\underline{\ WriteParagraphC\ } \\
\Xi EPC \\
output! : OUTPUT \\
\hline
\exists pC : PARAGRAPHC \mid \\
\qquad (1..totalC \lhd wordsC)\ evenPC\ pC \bullet \\
\qquad\qquad\qquad output! = formOutputC(pC) \\
\hline
\end{array}$$

6.4 Retrieve Relation and Proof Obligations

The retrieve relation is given in the schema *Retr*. It uses a function *map* that takes another function and a sequence and applies the function to every element of that sequence.

$$\begin{array}{l}
[X, Y] \\
\hline
map : (X \rightarrow Y) \rightarrow \mathsf{seq}\, X \rightarrow \mathsf{seq}\, Y \\
\hline
\forall f : X \rightarrow Y;\ xs : \mathsf{seq}\, S \bullet \\
\qquad map\ f\ \langle\rangle = \langle\rangle \wedge \\
\qquad map\ f\ xs = \langle f(head\ xs)\rangle ^\frown map\ f\ (tail\ xs) \\
\hline
\end{array}$$

$$\begin{array}{l}
\underline{\ Retr\ } \\
EP \\
EPC \\
\hline
words = map\ getWordC\ (1..totalC \lhd wordsC) \\
\hline
\end{array}$$

It is not difficult to see that the retrieve relation is functional. Hence, we may use the proof obligations for functional retrieve relations. The proof obligation for initial states is easy, and since the preconditions of the concrete operations are true, the proof obligations for applicability are trivially satisfied as well. Below, we sketch the correctness proof for $WriteParagraphC$. The correctness proof for $ReadInputC$ is similar.

6.4.1 Correctness Proof for $WriteParagraphC$

The first step in this proof is to prove theorems that relate the abstract and concrete functions. These theorems are given below. The details in their proofs are omitted, as these proofs are not difficult.

Theorem 6.1

$$\begin{array}{l} \forall\, ws : \mathrm{seq}\ WORD;\ wCs : \mathrm{seq}\ WordC \mid \\ \quad ws = map\ getWordC\ wCs \bullet \\ \qquad width(ws) = widthC(wCs) \end{array}$$

□

Theorem 6.2

$$\begin{array}{l} \forall\, p : PARAGRAPH;\ pC : PARAGRAPHC \mid \\ \quad p = map\ (map\ getWordC)\ pC \bullet \\ \qquad waste(p) = wasteC(pC) \end{array}$$

Proof: Use Theorem 6.1

□

Theorem 6.3

$$\begin{aligned}&\forall\, ws : \mathrm{seq}\ \mathit{WORD};\ \ wCs : \mathrm{seq}\ \mathit{WordC} \mid\\ &\quad ws = \mathit{map}\ \mathit{getWordC}\ \ wCs \bullet\\ &\qquad \mathit{minWaste}(ws) = \mathit{minWasteC}(wCs)\end{aligned}$$

Proof: Use Theorem 6.2

□

Theorem 6.4

$$\begin{aligned}&\forall\, ws : \mathrm{seq}\ \mathit{WORD};\ \ wCs : \mathrm{seq}\ \mathit{WordC};\\ &\ \ p : \mathit{PARAGRAPH};\ pC : \mathit{PARAGRAPHC} \mid\\ &\quad ws = \mathit{map}\ \mathit{getWordC}\ \ wCs \wedge p = \mathit{map}\ (\mathit{map}\ \mathit{getWordC})\ pC \bullet\\ &\qquad wCs\ \mathit{evenPC}\ pC \ \ \Rightarrow\ \ ws\ \mathit{evenP}\ p\end{aligned}$$

Proof: Use Theorem 6.2 and 6.3.

□

Theorem 6.5

$$\begin{aligned}&\forall\, l : \mathit{LINE};\ lC : \mathit{LINEC} \mid\\ &\quad l = \mathit{map}\ \mathit{getWordC}\ lC \bullet\\ &\qquad \mathit{insertSC}(lC) = \mathit{insertS}(l)\end{aligned}$$

Proof: By induction.

□

Theorem 6.6

$$\begin{array}{l}\forall\, p : PARAGRAPH;\ pC : PARAGRAPHC \mid \\ \quad p = map\ (map\ getWordC)\ pC \bullet \\ \qquad formOutputC(pC) = formOutput(p)\end{array}$$

Proof: By induction using Theorem 6.5.

□

<u>A Sketch of the Proof</u>

The correctness proof requirement is

$$\begin{array}{l}\forall\, EP;\ EP';\ EPC;\ EPC';\ output! : OUTPUT \bullet \\ \quad \text{pre } WriteParagraph \wedge Retr \wedge WriteParagraphC \wedge Retr' \\ \qquad \Rightarrow WriteParagraph.\end{array}$$

From the premise, we deduce

$$\begin{array}{l}\Rightarrow\ words = map\ getWordC\ (1..totalC \lhd wordsC) \wedge \\ \quad \exists\, pC : PARAGRAPHC \bullet \\ \qquad (1..totalC \lhd wordsC)\ evenPC\ pC \wedge \\ \qquad output! = formOutputC(pC)\end{array}$$

For every concrete paragraph, we can always find an abstract paragraph that has the same words. We existentially introduce this abstract paragraph into the predicate.

$$\begin{array}{l}\Rightarrow\ words = map\ getWordC\ (1..totalC \lhd wordsC) \wedge \\ \quad \exists\, pC : PARAGRAPHC;\ p : PARAGRAPH \bullet \\ \qquad p = map\ (map\ getWordC)\ pC \wedge \\ \qquad (1..totalC \lhd wordsC)\ evenPC\ pC \wedge \\ \qquad output! = formOutputC(pC)\end{array}$$

Using Theorem 6.4, the expression $(1..totalC \lhd wordsC)\ evenPC\ pC$ implies expression $words\ evenP\ p$.

$$
\begin{array}{ll}
\Rightarrow & \exists\, pC : PARAGRAPHC;\ p : PARAGRAPH \bullet \\
& \quad p = map\ (map\ getWordC)\ pC \wedge \\
& \quad words\ evenP\ p \wedge \\
& \quad output! = formOutputC(pC)
\end{array}
$$

Using Theorem 6.6, $formOutputC(pC)$ may be replaced by the $formOutput(p)$.

$$
\begin{array}{ll}
\Rightarrow & \exists\, pC : PARAGRAPHC;\ p : PARAGRAPH \bullet \\
& \quad words\ evenP\ p \wedge \\
& \quad output! = formOutput(p)
\end{array}
$$

Since pC is free, the existential quantification of pC may be removed, which completes our proof.

$$
\begin{array}{ll}
\Leftrightarrow & \exists\, p : PARAGRAPH \bullet \\
& \quad words\ evenP\ p \wedge \\
& \quad output! = formOutput(p)
\end{array}
$$

6.5 Using Predefined Pascal Routines

If the concrete operation of the previous section were to be translated, they would result in procedures with formal parameters *input?* and *output!*. These parameters may not be used because input and output streams are not system variables in Pascal and as such, cannot to be passed as parameters in a procedure call. Below, we view the input and output streams as state variables and modify the concrete operations appropriately to make use of them.

$$
\begin{array}{|l}
\textit{ReadInputC} \\
\hline
\Delta EPC \\
input, input' : INPUT \\
output, output' : OUTPUT \\
\hline
totalC'' = \\
\quad min \{ \\
\qquad min \{ n : \mathbb{N} \mid conSC(conWS \circ conSC)^n(input) = \langle\rangle \}, \\
\qquad maxWord \\
\qquad \} \\
\forall i : 1..totalC'' \bullet \\
\quad wordsC''(i).length = \\
\qquad \#(1..maxLength \lhd \\
\qquad\quad retWC(conSC((conWC \circ conSC)^{i-1}(input)))) \land \\
\quad wordsC''(i).word = wordsC(i).word \oplus \\
\qquad (1..maxLength \lhd \\
\qquad\quad retWC(conSC((conWC \circ conSC)^{i-1}(input)))) \\
\\
output' = output \\
\hline
\end{array}
$$

Instead of requiring *ReadInputC* to use the input variable *input?*, it is now required to use the input stream as the input. The operation *WriteParagraphC* is required to concatenate its output onto the output stream.

$$
\begin{array}{|l}
\textit{WriteParagraphC} \\
\hline
\Xi EPC \\
input, input' : INPUT \\
output, output' : OUTPUT \\
\hline
\exists pC : PARAGRAPHC \mid \\
\qquad (1..totalC \lhd wordsC) \; evenPC \; pC \bullet \\
\qquad\qquad\qquad output' = output \frown formOutputC(pC) \\
\\
input' = input \\
\hline
\end{array}
$$

Variable	Abbreviation
wordsC	*w*
totalC	*t*
input	*in*
output	*out*

Table 6.1: Abbreviations for the state variables of operations *ReadInputC* and *WriteParagraph*.

Using Pascal Input and Output Routines

Since we must manipulate the input and output streams through Pascal input and output routines, a way to introduce these routines into the development would be to specify them as procedures in our abstract program. By refining our operations to use these routines, we can provide a formal justification for their use.

Below, we give specifications for a few Pascal input and output routines. Since these specifications will be used in the refinement of our operations, it would be convenient to use the abbreviated form of the state variables. The Pascal routine

```
read
```

allows us to read a character from the input stream. A specification of this routine is contained in the procedure *read* below.

$$\begin{array}{l}\textbf{procedure}\ read(\textbf{value result}\ c : CHAR) \mathrel{\hat=} \\ \quad in, c : [in \neq \langle\rangle\ ,\ c = head\ in_0 \wedge in = tail\ in_0]\end{array}$$

A specification for the Pascal routine

```
write
```

which allows us to output one character (except for newline) is given in procedure *write*.

> **procedure** *write*(value $c : CHAR$) $\mathrel{\hat=}$
> $\quad out : [c \neq \mathsf{newline}\ ,\ out = out_0 \frown \langle c \rangle]$

A specification for the Pascal routine

```
writeln
```

which allows us to output a newline character is given in procedure *writeln*.

> **procedure** *writeln* $\mathrel{\hat=}$
> $\quad out : [true\ ,\ out = out_0 \frown \langle \mathsf{newline} \rangle]$

By declaring a character array as a *packed* array, we may make use of the Pascal routine that allows a prefix of the items in the array to be output. As an example, for a packed array a and an integer l, the Pascal command

`write(`$a : l$`)`

will output the first l characters of a. A specification for this Pascal command is given as procedure *writeArray*.

> **procedure** *writeArray*
> $\quad$(value $a : CHARARRAY$; value $l : 1..maxLength$) $\mathrel{\hat=}$
> $\quad\quad out : [true\ ,\ out = out_0 \frown (1..l \lhd a)]$

6.6 Operation Refinement

The modified concrete operations in the previous section may now be translated into procedures and refined using the refinement calculus. Below, we describe only the refinement of *WriteParagraphC* which is the operation for computing and outputting even paragraphs. We omit the refinement of *ReadInputC*.

6.6.1 Computing Minimum Waste Array

We specify and refine a procedure that computes the minimum waste of all prefixes of the word sequence. This will be used in the refinement of the procedure that computes and outputs even paragraphs.

```
procedure $ComputeMinWasteArray$
    (value result $mwA : 1..maxWord \to \mathbb{Z}$) $\mathrel{\widehat=}$
$mwA : [t \geq 1\ ,$
                  $(\forall i : \mathbb{Z} \mid 1 \leq i \leq t \bullet mwA[i] = minWasteC(w[i \to t]))]$
```

We take the liberty of writing $w[k \to l]$ for the sequence that consists of the kth to the lth elements of the sequence w.

The Refinement Steps

The next few refinement steps set up an iteration that enables us to consider progressively larger prefixes.

```
⊑  $I \mathrel{\hat=} (\forall i : \mathbb{Z} \mid j \le i \le l \bullet$
                 $mwA[i] = minWasteC(w[i \to l]) \wedge 1 \le j \le l)$
   var $j : \mathbb{Z} \bullet$
   $j := l;$
   $mwA(l) := 0;$
   $j, mwA : [I\ ,\ I \wedge j = 1]$                                  $\lhd$

⊑  do $j \ne 1 \to$
       $j := j - 1;$
       $j, mwA : [I[j \backslash j+1] \wedge j+1 \ne 1\ ,\ I]$              $\lhd$
   od
```

The specification statement in the body of the iteration computes the minimum waste of the sequence consisting of the last $(l - j + 1)$ words and stores this value in $mwA(j)$. We introduce variable x for the computation of the minimum waste of $w[j \to l]$. The value of x at the end of the computation will be assigned to $mwA(j)$.

```
⊑  var $x : \mathbb{Z} \bullet$
   $x, j : [I[j \backslash j+1] \wedge j+1 \ne 1\ ,\ x = minWasteC(w[j \to l])];$      $\lhd$
   $mwA(j) := x;$
```

Strategy for Computing Minimum Wastes

We use a strategy that computes the minimum waste of a prefix based on the minimum wastes of smaller prefixes. For this, we rewrite our definition of minimum waste as follows.

$minWasteC(w[j \to l])$

$$= \ min\ \{pC : PARAGRAPHC \mid {}^\frown/\ pC = w[j \to t] \bullet wasteC(pC)\}$$

$$\begin{aligned} = \ min\ \{lC : LINEC;\ pC : PARAGRAPHC \\ \mid {}^\frown/(\langle lC\rangle {}^\frown pC) = w[j \to t] \\ \bullet\ wasteC(\langle lC\rangle {}^\frown pC)\}\end{aligned}$$

$$\begin{aligned} = \ min\ \{lC : LINEC;\ pC : PARAGRAPHC \\ \mid lC {}^\frown ({}^\frown/\ pC) = w[j \to t] \\ \bullet\ wasteC(\langle lC\rangle {}^\frown pC)\}\end{aligned}$$

$$= \ min \left\{ \begin{array}{l} k : \mathbb{Z} \\ pC : PARAGRAPH \\ \quad \left| \begin{array}{l} j \le k \le t \ \wedge \\ {}^\frown/\ pC = w[k+1 \to t] \ \wedge \\ \sum_{i=j}^{k} w(i).length + (k-j) \le maxLength \end{array} \right. \\ \qquad \bullet\ waste(\langle w[j \to k]\rangle {}^\frown pC)\} \end{array} \right. \qquad (*)$$

<u>Case 1</u>

We now have two cases. First, if the words of the prefix may all be laid out on one line, then the minimum waste is zero, since the last line of a paragraph does not contribute to the paragraph's waste.

$$\begin{aligned} & \textstyle\sum_{i=j}^{t} w(i).length + (t-j) \le maxLength \\ \Rightarrow\ & (*) = 0 \end{aligned}$$

<u>Case 2</u>

The second case is when the words of the prefix cannot be written as a one line paragraph.

$$\sum_{i=j}^{t} w(i).length + (t-j) > maxLength$$

$$\Rightarrow\ (*) = min \left\{ \begin{array}{l} k : \mathbb{Z} \\ pC : PARAGRAPHC \end{array} \right. \begin{array}{|l} j \leq k < t \wedge \\ {}^\frown/\, pC = w[k+1 \to t] \wedge \\ \sum_{i=j}^{k} w(i).length + (k-j) \leq maxLength \end{array} \bullet\ max \left\{ \begin{array}{l} maxLength - (k-j) - \\ \quad \sum_{i=j}^{k} w(i).length, \\ wastePC(pC) \end{array} \right\} \Bigg\}$$

We do not have to consider the case when $k = t$ since it is taken care by the first case.

$$= min \left\{ k : \mathbb{Z} \left| \begin{array}{l} j \leq k < t \wedge \\ \sum_{i=j}^{k} w(i).length + (k-j) \leq maxLength \end{array} \right. \bullet \right. \\ min \left\{ pC : PARAGRAPHC \mid {}^\frown/\, pC = w[k+1 \to t] \bullet \right. \\ \left. \left. max \left\{ \begin{array}{l} maxLength - (k-j) - \\ \quad \sum_{i=j}^{k} w(i).length, \\ wastePC(pC) \end{array} \right\} \right\} \right\}$$

$$= min \left\{ k : \mathbb{Z} \left| \begin{array}{l} j \leq k < t \wedge \\ \sum_{i=j}^{k} w(i).length + (k-j) \leq maxLength \end{array} \right. \bullet \right. \\ max \left\{ maxLength - (k-j) - \sum_{i=j}^{k} w(i).length, \right. \\ \left. \left. min \left\{ \begin{array}{l} pC : PARAGRAPHC \\ \quad \mid {}^\frown/\, pC = w[k+1 \to t] \\ \quad \bullet\ wastePC(pC) \end{array} \right\} \right\} \right\}$$

$$= min \left\{ k : \mathbb{Z} \left| \begin{array}{l} j \leq k < t \wedge \\ \sum_{i=j}^{k} w(i).length + (k-j) \leq maxLength \end{array} \right. \bullet \right. \\ \left. max \left\{ \begin{array}{l} maxLength - (k-j) - \sum_{i=j}^{k} w(i).length, \\ minWaste(w[k+1 \to t]) \end{array} \right\} \right\}$$

Refinement Continued

In the previous section, we defined the minimum waste of a prefix $w[j \rightarrow t]$ in terms of the minimum wastes of smaller prefixes $w[k+1 \rightarrow t]$. In the following, we continue the development of the program using this alternate definition of minimum waste.

$$
\begin{array}{ll}
\sqsubseteq & X \mathrel{\widehat{=}} min\ \{k : \mathbb{Z} \\
 & \qquad\qquad \left|\ \begin{array}{l} j \leq k < n \\ \sum_{i=j}^{k} w(i).length \leq maxLength \end{array}\right. \\
 & \qquad\qquad \bullet\ max \left\{ \begin{array}{c} maxLength - (k-j)- \\ \sum_{i=j}^{k} w(i).length, \\ minWaste(w[k+1 \rightarrow t]) \end{array} \right\} \Bigg\} \\
 & J \mathrel{\widehat{=}} I[j \backslash j+1] \wedge \\
 & \qquad j+1 \neq -1 \wedge \\
 & \qquad x = X \wedge \\
 & \qquad s = \sum_{i=j}^{n} w(i).length + (n-j) \wedge \\
 & \qquad j \leq n \leq t
\end{array}
$$

```
var $s, n : \mathbb{Z}$ •
$n := j + 1;$
$s := w(j).length + w(j+1).length + 1;$
$x := max(maxLength - w(j).length, mwA(j+1));$
$s, n, x : [J\ ,\ J \wedge (n = t \vee s > maxLength)];$          $\lhd$
if $s \leq maxLength \rightarrow s := 0$
[] $s > maxLength \rightarrow$ skip
fi
```

```
$\sqsubseteq$  do $(n \neq t \wedge s \leq maxLength) \rightarrow$
    $s, n, x : \left[ \begin{array}{l} n \neq t \wedge \\ s \leq maxLength \wedge \\ J \end{array} ,\ \begin{array}{c} J \wedge \\ 0 \leq t - n \leq t - n_0 \end{array} \right]$          $\lhd$
od
```

$$\begin{array}{ll}\sqsubseteq & x := min(x, max(s, mwA(n+1)));\\ & s := s + w(n+1).length + 1;\\ & n := n + 1\end{array}$$

In the preceding steps, we have assumed the availability of functions *max* and *min* in the Pascal programming language. Although these functions are not available in Pascal, their correct constructions are easy. The code from the above refinement is collected in Figure 6.3.

6.6.2 Writing a Line

In Figure 6.4, we give a specification and code for a procedure that outputs one line of a paragraph. This procedure will be used in the development of the next section. Its refinement is not difficult and is omitted. Notice that this procedure uses some of the system routines of Section 6.5.

6.6.3 Writing an Even Paragraph

We specify and refine a procedure that computes an even paragraph. This procedure uses the minimum waste array that is computed in Section 6.6.1.

$$\begin{array}{l}\textbf{procedure } WriteEven\\ \quad (\textbf{value } mwA : 1..maxWord \rightarrow \mathbb{Z}) \mathrel{\widehat{=}}\\ out : [\forall i : 1..t \mid mwA(i) = min\,WasteC(w[i \rightarrow t]) \wedge t \geq 1\ ,\\ \qquad \left.\begin{array}{l}\exists pC : PARAGRAPHC \mid\\ \quad (1..t \lhd w)\ evenPC\ pC \bullet\\ \qquad out = out_0 \frown formOutputC(pC)\end{array}\right] \qquad \lhd\end{array}$$

```
    procedure $ComputeMinWasteArray$
        (value result $mwA : 1..maxWord \rightarrow \mathbb{Z}$) $\mathrel{\widehat=}$
    $mwA : [t \geq 1\ ,$
        $(\forall\, i : \mathbb{Z} \mid 1 \leq i \leq t \bullet mwA[i] = minWaste(w[i \rightarrow t]))]$
$\sqsubseteq$  $[\![$ var $j, n, s, x : \mathbb{Z} \bullet$
        $j := t;$
        $mwA(t) := 0;$
        do $j \neq i \rightarrow$
            $j := j - 1;$
            $n := j + 1;$
            $s := w(j).length + w(j+1).length + 1;$
            $x := max\ (maxLength - w(j).length, mwA(j+1));$
            do $n \neq t \wedge s \leq maxLength \rightarrow$
                $x := min\ (x, max\ (maxLength - s, mwA(n+1)));$
                $s := s + w(n+1).length + 1;$
                $n := n + 1$
            od;
            if $s \leq maxLength \rightarrow$
                $x := 0$
            $[\!]$ $s > maxLength \rightarrow$
                skip
            fi;
            $mwA(j) := x$
        od;
    $]\!]$
```

Figure 6.3: A possible refinement of the procedure *ComputeMinWasteArray*.

```
    procedure $WriteLine$(value $s, f : \mathbb{Z}$) $\triangleq$
    $out : [true\ ,\ out = out_0 \frown insertSC(w[s \rightarrow f]) \frown \langle \mathsf{newline} \rangle]$
$\sqsubseteq$   $[\![$ var $k : \mathbb{Z} \bullet$
        $writeArray(w(s).word, w(s).length)$;
        $k := s + 1$;
        do $k \leq f \rightarrow$
                $write(\mathsf{space})$;
                $writeArray(w(k).word, w(k).length)$;
                $k := k + 1$
        od;
        $writeln$
    $]\!]$
```

Figure 6.4: A possible refinement of the procedure *WriteLine*.

The Refinement Steps

For procedure *WriteEven*, we use a strategy that outputs even paragraphs line by line. For this, an iteration is set up where the variable i refers to the first word of the current line being printed.

```
$\sqsubseteq$   $A \triangleq (\forall i : 1..t \mid mwA(i) = minWaste(w[i \rightarrow t]) \wedge t \geq 1)$
    $I \triangleq \exists p, q : PARAGRAPHC \bullet$
            $w[1 \rightarrow i-1]\ evenPC\ p\ \wedge$
            $w[i \rightarrow t]\ evenPC\ q\ \wedge$
            $w\ evenPC\ p \frown q\ \wedge$
            $out = OUT \frown formOutputC(q)$
    con $OUT$
    var $i : \mathbb{Z} \bullet$
        $i := 1$;
        $i, out : [I \wedge A\ ,\ I \wedge A \wedge i = t + 1]$                    $\triangleleft$
```

```
$\sqsubseteq$  do $i \neq t+1$
         $i, out : [i \neq t+1 \wedge I \wedge A \ , \ I \wedge A \wedge 0 \leq t-i < t-i_0]$    $\lhd$
     od
```

A variable j is used to find the end of the current line. If both the waste of the current line $w[i \rightarrow j]$ and the minimum waste of the remaining sequence $w[j+1 \rightarrow t]$ are each less than the minimum waste of the whole sequence of word, we may take $w[i \rightarrow j]$ as a legal line of the even paragraph.

```
$\sqsubseteq$  $J \mathrel{\hat=} \exists p, q : PARAGRAPHC \bullet$
         $w[1 \rightarrow i-1]\ evenPC\ p\ \wedge$
         $w[i \rightarrow t]\ evenPC\ q\ \wedge$
         $w\ evenPC\ p \frown q\ \wedge$
         $i \leq j \leq t\ \wedge$
         $w[i \rightarrow j]\ suffix\ q\ \wedge$
         $maxLength - widthC(w[i \rightarrow j]) \geq 0$
     var $j : \mathbb{Z} \bullet$
     $j : [i \neq t+1 \wedge I \wedge A \ ,$
         $A \wedge J \wedge$
         $((maxLength - widthC(w[i \rightarrow j]) \leq minWaste([1 \rightarrow t]) \wedge$
         $minWaste(w[j+1 \rightarrow t]) \leq minWaste(w[1 \rightarrow t])) \vee j = t)\ ]$ ; $\lhd$
     $WriteLine(i, j)$;
     $i := j+1$

$\sqsubseteq$  $K \mathrel{\hat=} J \wedge s = maxLength - widthC(w[i \rightarrow j])$
     var $s \bullet$
     $j := i$;
     $s := maxLength - w(j).length$;
     $j : [A \wedge K \ ,$
         $A \wedge K \wedge$
         $((s \leq minWaste(w[1 \rightarrow t]) \wedge$
         $minWaste(w[j+1 \rightarrow t]) \leq minWaste(w[1 \rightarrow t])) \vee j = t)\ ]$ $\lhd$
```

```
    procedure $WriteEven$
        (value $mwA : 1..maxWord \to \mathbb{Z}) \mathrel{\widehat{=}}$
    $out : [\forall i : 1..t \mid mwA(i) = minWasteC(w[i \to t]) \wedge t \geq 1 \bullet$
            $\exists pC : PARAGRAPHC \mid$
                $(1..t \lhd w) evenPCpC \bullet$
                    $out = out_0 \frown formOutputC(pC)]$
$\sqsubseteq$   |[ var $i, j, s : \mathbb{Z} \bullet$
        $i := 1;$
        do $i \neq t + 1 \to$
            $j := i;$
            $s := maxLength - w(j).length;$
            do $(j \neq t) \wedge ((s > mwA(1)) \vee$
                        $(mwA(j + 1) \leq mwA(1))) \to$
                    $j := j + 1;$
                    $s := s - w(j).length - 1$
            od;
            $WriteLine(i, j);$
            $i := j + 1$
        od
    ]|
```

Figure 6.5: Code from the refinement of procedure *WriteEven*.

```
$\sqsubseteq$   do $(j \neq t) \wedge \left( \begin{array}{l} s > mwA(1) \vee \\ mwA(j+1) \leq mwA(1) \end{array} \right) \to$
        $j := j + 1;$
        $s := s - w(j).length - 1$
    od
```

Collecting all code from the development of this section, we have the refined procedure of Figure 6.5.

```
    procedure $WriteParagraphC \mathrel{\widehat{=}}$
    $out : [true\ ,$
                $\exists pC : PARAGRAPHC \mid$
                    $(1..t \lhd w)\ evenPC\ pC \bullet$
                        $out = out_0 \frown formOutputC(pC)\ ]$
$\sqsubseteq$   $[\![$ var $mwA : 1..maxWord \to \mathbb{Z} \bullet$
        if $t \geq 1 \to$
                $ComputeMinWasteArray(mwA)$;
                $WriteEven(mwA)$
        $[\!]\ t = 0 \to$
                skip
        fi
    $]\!]$
```

Figure 6.6: A refinement of procedure *WriteParagraphC* that uses procedures *ComputeMinWasteArray* and *WriteEven*.

6.6.4 Computing an Even Paragraph

The procedure *WriteParagraphC* for computing and outputting even paragraphs is given in Figure 6.6. It makes use of the procedures that are developed in the earlier parts of this section. Again, we omit its refinement since it is not difficult.

6.7 Summary

In this chapter, we have sketched the development of a program that computes even paragraphs. This problem was specified by Bird in [5], where he also developed a program in a functional language to solve it. Morgan specified a simplified version of the same problem in the refinement calculus and outlined a solution where paragraphs were abstracted as sequences containing sequences of word lengths [31]. Our work here is more pragmatic and complete than Morgan's since we consider a word as a sequence of characters and develop a Pascal program to solve the problem. This program is given in Appendix C.

Chapter 7

Concluding Remarks

In this thesis, we have studied a formal software development process that uses the formal specification language called Z, and the formal development method called the refinement calculus. Z is suitable for specification since its schema calculus and mathematical toolkit allow large and complex systems to be described modularly and compactly. The refinement calculus is appropriate for development since its notation allows executable and non-executable constructs to be treated in the same framework.

The software development process is be divided into five stages: formal specification in Z, data refinement, translation into the refinement calculus, operation refinement, and translation into the target programming language. In this thesis, we have collected together and illustrated many of the important results for understanding and using this process. In particular, we have shown, by examples, how a software system may be developed *all the way* from specification to program.

7.1 Directions for Further Research

Below, we give some suggestions and directions for further research.

7.1.1 System Development Tool Support

As demonstrated in the earlier chapters of this thesis, the amount of mathematical activity needed for a formal development can be quite enormous, especially for large and complex systems. We feel that much of this activity may be less difficult to accomplish if support tools are available. Below, we give some indication of the desired properties of these tools.

Formal Specification and Data Refinement

Obviously, it would be advantageous to have tools to edit, format and typecheck Z specifications. Some tools that provide these features may be found in the catalogue compiled by Parker [38]. Since Z specifications can get very large and complex, it would be beneficial to have a tool that manages schemas. A visual editor that allows the interactive editing, storing, organizing and retrieval of schemas would definitely ease the reading and writing of specifications for large and complex systems.

Although there are ways to organize the proof obligations based on the structure of a specification and its concrete design, the amount of effort needed to manage these proofs can be formidable. As such, a tool that does at least "housekeeping" of the proof steps would be of great help. Several such proof tools have been used with Z. Some of these are described in [1, 36, 37, 39].

Translation into the Refinement Calculus and Operation Refinement

Since Z has a well-defined syntax, it may be possible to have tools to assist the translation from a concrete design into the refinement calculus. A more difficult requirement would be an environment where refinement may be carried out interactively. Similar to the "housekeeping" problem of proofs in Z, refinement steps in a development may be numerous and elaborate. A tool that manages these steps must allow the user to easily copy, delete, and insert predicates. Furthermore, it would be useful to have some mechanism by which the refinement steps may be automatically checked against the refinement laws.

7.1.2 Libraries of Specifications and Refinements for Data Structures

Since it is common to build large systems out of standard data structures, it would be useful to have a library of specifications and refinements for common data structures. A formal specification or concrete design of a system may use these specifications from the library simply by renaming the appropriate components of the schemas. When the specification or concrete design of the system is finally translated into the refinement calculus, the resulting abstract program may be refined to introduce the procedures of these data structures whose refinements are already present in the library. Such a library would provide opportunities for reuse.

7.1.3 Calculating Data Refinement

As mentioned in the last section of Chapter 3, there is a technique of data refinement where a concrete operation may be calculated directly from its abstract specification and the retrieve relation [21, 22, 45]. Due to the calculative nature of the refinement calculus, this method of data refinement may be more appropriate for our purpose since it would enable our development process to be viewed as a more uniform method.

7.1.4 Translation Rules for Other Z Constructs

In our exposition on the translation from Z to the refinement calculus, we have given several rules for translating operation schemas directly into executable structures based on the way that they are connected by schema connectives. A direction for further research would be to discover executable constructs to translate other Z structures. For example, it may be worthwhile to design similar transformation rules for sequential composition and piping in Z.

An inflexibility that we have noticed in our translation scheme is that input variables and output variables of an operation schema are given value and result substitutions in the resulting procedure. This may be too restrictive especially when a substitution method is not available in the target programming language. Although it is possible to change the substitution of a formal parameter within the framework of the refinement calculus, it is more convenient to have the freedom to choose the appropriate substitution method during the translation stage. As such, it would be helpful to formulate rules regarding how substitution methods may be used during the stage of translation.

7.1.5 Data Refinement in the Refinement Calculus

Although King advocated that the task of data refinement be performed before the translation into the refinement calculus, he also indicated the possibility of delaying data refinement until after the notational change from Z to the refinement calculus [25]. This approach would involve the use of the data refinement techniques that are present in the refinement calculus [34, 28, 27]. A point of research here would be to explore the advantages and disadvantages of such an approach.

7.1.6 Operation Refinement for Dynamic Data Structures

In this thesis, we have restricted ourselves to *static* data structures like integers, characters and fixed-length arrays. Our experiments with pointers have shown that it could be difficult to refine programs with *dynamic* data structures. Although lists and trees are easier than pointers when used for program derivation, the study of pointers should not be ignored since they are efficient and are commonly used to implement types like lists and trees. As such, it would be worthwhile to formulate mathematical models and laws for using pointers in the refinement calculus. We point the reader to [4] for a discussion on calculating programs with pointers.

Bibliography

[1] R.D. Arthan. On formal specification of a proof tool. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 356–370. Springer-Verlag, 1991.

[2] R. J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[3] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[4] A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–205, Semtember 1989.

[5] R.S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6:159–189, 1986.

[6] Jonathan P. Bowen. Formal specification in Z as a design and documentation tool. In *Proc. Second IEE/BCS Conference on Software Engineering*, volume 290, pages 164–168. IEE/BCS, July 1988.

[7] Jonathan P. Bowen. Formal specification of window systems. Technical Monograph PRG-74, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, June 1989.

[8] Jonathan P. Bowen. Selected Z bibliography. Oxford University Computing Laboratory, 1990–1992.

[9] Jonathan P. Bowen. X: Why Z? *Computer Graphics Forum*, 11(4):221–234, 1992.

[10] Doug Cooper. *Condensed Pascal.* W.W. Norton, New York and London, 1987.

[11] Norman Delisle and David Garlan. Formally specifying electronic instruments. In *Proc Fifth International Workshop on Software Specification and*

Design. IEEE Computer Society, May 1989. Also published in ACM SIG-SOFT Software Engineering Notes 14(3).

[12] Norman Delisle and David Garlan. A formal specification of an oscilloscope. *IEEE Software*, pages 29–36, September 1990.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.

[14] Veronika Doma and Robin Nicholl. EZ: A system for automatic prototyping of Z specifications. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 1991.

[15] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[16] Patrick A.V. Hall. Towards testing with respect to formal specification. In *Proc. Second IEE/BCS Conference on Software Engineering*, volume 290, pages 159–163, Liverpool, UK, July 1988, July 1988. IEE/BCS.

[17] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(1):330–338, November 1989.

[18] Ian J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, January 1986.

[19] Ian J. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd edition, 1992.

[20] He, Jifeng, C.A.R. Hoare, and Jeff W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

[21] C.A.R. Hoare, He, Jifeng, and Jeff W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, 1987.

[22] Mark B. Josephs. The data refinement calculator for Z specifications. *Information Processing Letters*, 27(1):29–33, 1988.

[23] P. King. Prototyping Z specifications. In G. Rose and I. Hayes, editors, *Second Half-yearly Report, UQ/QTC Collaborative Programme in Formal Description Techniques*, pages 5.1 –5.23. Dept. of Computer Science, University of Queensland, February 1988.

[24] S. King and I.H. Sorensen. From specification, through design to code: A case study in refinement. In P.N. Scharbach, editor, *Formal Methods: Theory and Practice*, pages 103–137. Oxford, 1989.

[25] Steve King. Z and the refinement calculus. In Dines Bjørner, C.A.R. Hoare, and Hans Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 164–188. VDM-Europe, Springer-Verlag, 1990.

[26] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.

[27] C. Carroll Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 29(6):293–296, December 1988.

[28] C. Carroll Morgan. Data refinement using miracles. *Information Processing Letters*, 26(5):243–246, January 1988.

[29] C. Carroll Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1), October 1988.

[30] C. Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3), July 1988.

[31] C. Carroll Morgan. *Programming from Specifications.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1990.

[32] C. Carroll Morgan. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14(2-3):281–304, October 1990.

[33] C. Carroll Morgan and Ken A. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5), September 1987.

[34] Carroll Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[35] J. M. Morris. Programs from specifications. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs.* Addison-Wesley, 1989.

[36] David S. Neilson. Machine support for Z: the zedB tool. In John E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 105–128. Springer-Verlag, 1991.

[37] David S. Neilson and Divya Prasad. zedB : A proof tool for Z built on B. In John E. Nicholls, editor, *Z User Workshop, Oxford 1991*, Workshops in Computing, pages 243–258. Springer-Verlag, 1992.

[38] Colin E. Parker. Z tools catalogue. Zip document zip/bae/90/020, British Aerospace, Warton, May 1991.

[39] Mark Saaltink. Z and Eves. In John E. Nicholls, editor, *Z User Workshop, Oxford 1991*, Workshops in Computing, pages 211–232. Springer-Verlag, 1992.

[40] J. Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, January 1988.

[41] J. Michael Spivey. Specifying a real-time kernel. *IEEE Software*, pages 21–28, September 1990.

[42] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd edition, 1992.

[43] Susan Stepney and Rosalind Barden. Annotated Z bibliography. *Bulletin of the EATCS*, (50):280–313, June 1993.

[44] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[45] J.C.P. Woodcock. A tutorial on the refinement calculus. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 79–140. VDM-Europe, Springer-Verlag, 1991.

[46] Jim C.P. Woodcock. Calculating properties of Z specifications. *ACM SIG-SOFT Software Engineering Notes*, 14(4):43-54, 1989.

[47] J.B. Wordsworth. *Software Development With Z*. Addison-Wesley, Workingham, England, 1992.

Appendix A

A Glossary of Z Notation

A glossary of the Z notation is given here for easy reference. The material here is compiled from [40, 42, 18].

A.1 Logic

$\neg P$	Not P.
$P \wedge Q$	P or Q.
$P \vee Q$	P or Q.
$P \Rightarrow Q$	P implies Q.
$P \Leftrightarrow Q$	P if and only if Q.
$\forall x : T \bullet Q$	For all x of type T, x satisfies Q.
$\forall x : T \mid P \bullet Q$	For all x of type T that satisfies P, x satisfies also Q.
	$\forall x : T \mid P \bullet Q \mathrel{\widehat{=}} (\forall x : T \bullet P \Rightarrow Q)$.
$\exists x : T \bullet Q$	There exists an x of type T that satisfies Q.
$\exists x : T \mid P \bullet Q$	There exists an x of type T that satisfies both P and Q.
	$\exists x : T \mid P \bullet Q \mathrel{\widehat{=}} (\exists x : T \bullet P \wedge Q)$.

A.2 Sets

$x \in S$	x is a member of S.
$S \subseteq T$	S is a subset of T.
$\varnothing$	The empty set.
$\{x_1, \ldots, x_n\}$	The set containing exactly $x_1, \ldots, x_n$.
$\{ x : T \mid P \}$	The set containing those x of type T which satisfy P.
$\{ x : T \mid P \bullet t \}$	The set of values of t for those x of type T satisfying P.
$(x_1, \ldots, x_n)$	Ordered n-tuple.

$S_1 \times \cdots \times S_n$	Cartesian product.
$\mathbb{P} S$	The set of all subsets of S.
$S \cap T$	Intersection of S and T.
$S \cup T$	Union of S and T.
$S \setminus T$	Set difference.
$\#S$	Size of finite set S.
$\mathbb{N}$	The natural numbers, $\{0, 1, 2, \ldots\}$.
$\mathbb{Z}$	The integers.
$m..n$	The range m up to n.
	$\mathrel{\widehat=} \{k : \mathbb{N} \mid m \leq k \land k \leq n\}$

A.3 Relations

$X \leftrightarrow Y$	Binary relations between X and Y.
	$\mathrel{\widehat=} \mathbb{P}(X \times Y)$.
$x \; R \; y$	x and y are related by R.
	$\mathrel{\widehat=} (x, y) \in R$
$x \mapsto y$	'Maplet' from x to y.
	$\mathrel{\widehat=} (x, y)$
$\operatorname{dom} R$	Domain of R.
	$\mathrel{\widehat=} \{x : X \mid (\exists y : Y \bullet x \; R \; y)\}$
$\operatorname{ran} R$	Range of R.
	$\mathrel{\widehat=} \{y : Y \mid (\exists x : X \bullet x \; R \; y)\}$
$R_1 \circ R_2$	Composition of relations.
	$\mathrel{\widehat=} \{x : X; \; z : Z \mid (\exists : Y \bullet x \; R_2 \; y \land y \; R_1 \; z)\}$
$R(\!\vert S \vert\!)$	Relational image.
	$\mathrel{\widehat=} \{y : Y \mid (\exists x : S \bullet x \; R \; y)\}$
$S \lhd R$	Domain restriction.
	$\mathrel{\widehat=} \{x : X; \; y : Y \mid x \in S \land x \; R \; y\}$
$R \rhd T$	Range restriction.
	$\mathrel{\widehat=} \{x : X; \; y : Y \mid x \; R \; y \land y \in T\}$

A.4 Functions

$X \nrightarrow Y$	Partial functions from X to Y.
	$\mathrel{\widehat=} \{f : X \leftrightarrow Y \mid f \circ f^{-1} \subseteq \operatorname{id} Y\}$
$X \rightarrow Y$	Total functions from X to Y.
	$\mathrel{\widehat=} \{f : X \nrightarrow Y \mid \operatorname{dom} f = X\}$
$X \twoheadrightarrow Y$	Finite partial functions from X to Y.

	$\mathrel{\widehat=} \{f : X \mathrel{+\!\!\!\!\twoheadrightarrow} Y \mid \mathrm{dom}\, f \in \mathbb{F}\, X\}$
$X \rightarrowtail\!\!\!\!\!+ \; Y$	Partial injections from X to Y.
	$\mathrel{\widehat=} \{f : X \mathrel{+\!\!\!\!\twoheadrightarrow} Y \mid f^{-1} \in Y \mathrel{+\!\!\!\!\twoheadrightarrow} X\}$
$X \rightarrowtail Y$	Total injections from X to Y.
	$\mathrel{\widehat=} (X \rightarrowtail\!\!\!\!\!+ \; Y) \cap (X \rightarrow Y)$
$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$	Bijections from X to Y.
	$\mathrel{\widehat=} \{f : X \rightarrowtail Y \mid \mathrm{ran}\, f = Y\}$
$f\ x, f(x)$	Function f applied to argument'x.
	$f\ x\ y \mathrel{\widehat=} (f\ x) y$.
$f \oplus g$	Functional overriding.
	$\mathrel{\widehat=} ((X \setminus \mathrm{dom}\ g) \lhd f) \cup g$

A.5 Sequences

$\mathrm{seq}\, X$	Sequences over X.
	$\mathrel{\widehat=} \{s : \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} X \mid \mathrm{dom} s = 1..\#s\}$
$\#s$	Length of s.
$\langle\rangle$	Empty sequence.
	$\mathrel{\widehat=} \varnothing$
$\langle x_1, \ldots, x_n\rangle$	The sequence containing $x_1, ..., x_n$.
	$\mathrel{\widehat=} \{1 \mapsto x_1, \ldots, n \mapsto n_x\}$
$s \frown t$	Concatenation of s and t.

Appendix B

Some Definitions, Abbreviations, and Laws of the Refinement Calculus

Below are some definitions, abbreviations, and laws of the refinement calculus. These are part of a more complete list which may be found in [31, pp. 227-240].

B.1 Definitions

B.1.1 Feasibility

Definition B.1 (feasibility "feas") The specification $w : [pre\ ,\ post]$ is *feasible* if and only if

$$(w = w_0) \wedge pre \wedge inv \ \Rightarrow\ (\exists\, w : T \bullet inv \wedge post),$$

where T is the type of w, and inv is the invariant that is associated with the variables w during their declarations.
□

B.2 Abbreviations

Abbreviation B.1 (initial variable "iv") Occurrences of 0-subscripted variables in the postcondition of a specification refer to values held by those variables in the *initial* state. Let x be any variable, probably occurring in the frame w. If X is a fresh name, and T is the type of x, then

$$\begin{array}{l} w : [pre\ ,\ post] \\ \mathrel{\widehat{=}}\ |[\ \mathbf{con}\ X : T \bullet w : [pre \wedge x = X\ ,\ post[x_0 \backslash X]]\ ]|. \end{array}$$

We reserve 0-subscripted names for that purpose, and call them *initial variables.*
□

Abbreviation B.2 (assumption "assum")

$$\{pre\}\ \mathrel{\widehat{=}}\ : [pre\ ,\ true].$$

□

Abbreviation B.3 (coercion "Coerc")

$$[post]\ \mathrel{\widehat{=}}\ : [true\ ,\ post].$$

□

Abbreviation B.4 (specification invariant "si")

$$w : [pre\ ,\ inv\ ,\ post]\ \mathrel{\widehat{=}}\ w : [pre \wedge inv\ ,\ inv \wedge post].$$

□

B.3 Laws

B.3.1 Assumption and Coercion

Law B.1 (introduce assumption "ia")

$$[post]\ \sqsubseteq\ [post]\ \{post\}.$$

□

Law B.2 (introduce coercion "ic") The program skip is refined by any coercion.

$$\mathbf{skip}\ \sqsubseteq\ [post].$$

□

Law B.3 (remove assumption "ra") Any assumption is refined by skip.

$$\{pre\} \quad \sqsubseteq \quad \textbf{skip}.$$

□

Law B.4 (remove coercion "rc")

$$\{pre\}\,[pre] \quad \sqsubseteq \quad \{pre\}.$$

□

Law B.5 (merge annotations "ma")

$$\begin{array}{lcl} \{pre'\}\,\{pre\} & = & \{pre' \wedge pre\} \\ [post]\,[post'] & = & [post \wedge post']. \end{array}$$

□

Law B.6 (absorb assumption "aa") An assumption before a specification can be absorbed directly into its precondition.

$$\begin{array}{ll} & \{pre'\};\ w : [pre\ ,\ post] \\ = & w : [pre' \wedge pre\ ,\ post]. \end{array}$$

□

Law B.7 (absorb coercion "ac") An coercion following a specification can be absorbed directly into its postcondition.

$$\begin{array}{ll} & w : [pre\ ,\ post];\ [post'] \\ = & w : [pre\ ,\ post \wedge post']. \end{array}$$

□

B.3.2 Pre- and Postcondition

Law B.8 (weaken precondition "wp") If $pre \Rightarrow pre'$, then

$$w : [pre \;,\; post] \quad \sqsubseteq \quad w : [pre' \;,\; post].$$

□

Law B.9 (strengthen postcondition "sp") If $pre[w \backslash w_0] \wedge post' \Rightarrow post$, then

$$w : [pre \;,\; post] \quad \sqsubseteq \quad w : [pre \;,\; post'].$$

□

B.3.3 Frame

Law B.10 (expand frame "efI")

$$w : [pre \;,\; post] \quad = \quad w, x : [pre \;,\; post \wedge x = x_0].$$

□

Law B.11 (expand frame "efII") For fresh constant X,

$$\begin{array}{ll} & w : [pre \;,\; post] \\ \sqsubseteq & \textbf{con } X \bullet \\ & w, x : [pre \;,\; post \wedge x = x_0]. \end{array}$$

□

Law B.12 (contract frame "cf")

$$w, x : [pre \;,\; post] \quad \sqsubseteq \quad w : [pre \;,\; post[x_0 \backslash x]].$$

□

B.3.4 Local Block

Law B.13 (introduce local block "ilb") If w and x are disjoint, then

$$w : [pre\ ,\ post] \quad \sqsubseteq \quad |[\ \mathsf{var}\ x : T;\ \mathsf{and}\ inv \bullet w, x : [pre\ ,\ post]\ ]|.$$

□

Law B.14 (local block initialization "lbi")

$$\begin{aligned} & |[\ \mathsf{var}\ l : T;\ \mathsf{initially}\ inv \bullet prog\ ]| \\ \sqsubseteq\ & |[\ \mathsf{var}\ l : T \bullet l : [true\ ,\ inv];\ prog\ ]|. \end{aligned}$$

□

B.3.5 Logical Constant

Law B.15 (introduce logical constant "ilc") If $pre \Rightarrow (\exists\, c : T \bullet pre')$, and c is a fresh name (it does not occur in w, pre, and $post$), then

$$\begin{aligned} & w : [pre\ ,\ post] \\ \sqsubseteq\ & \mathsf{con}\ c : T \bullet \\ & w : [pre'\ ,\ post]. \end{aligned}$$

□

Law B.16 (remove logical constant "rlc") If c occurs nowhere in program $prog$, then

$$|[\ \mathsf{con}\ c : T \bullet prog\ ]| \quad \sqsubseteq \quad prog.$$

□

Law B.17 (fix initial value "fiv") For any expression E such that $pre \Rightarrow E \in T$, and fresh name c,

$$\begin{aligned} & w : [pre\ ,\ post] \\ \sqsubseteq\ & \mathsf{con}\ c : T \bullet \\ & w : [pre \wedge c = E\ ,\ post]. \end{aligned}$$

□

B.3.6 Assignment

Law B.18 (simple specification "ss")

$$w := E \quad = \quad w : [true \ , \ w = E_0],$$

where E_0 is $E[w \backslash w_0]$.
□

Law B.19 (assignment "ass") If $(w = w_0) \wedge pre \Rightarrow post[w \backslash E]$, then

$$w, x : [pre \ , \ post] \quad \sqsubseteq \quad w := E.$$

□

Law B.20 (leading assignment "la") For any expression E,

$$\begin{aligned} & w, x : [pre[x \backslash E] \ , \ post[x_0 \backslash E_0]] \\ \sqsubseteq \quad & x := E; \\ & w, x : [pre \ , \ post]. \end{aligned}$$

The expression E_0 abbreviates $E[w, x \backslash w_0, x_0]$.
□

Law B.21 (following assignment "fa") For any expression E,

$$\begin{aligned} & w, x : [pre \ , \ post] \\ \sqsubseteq \quad & w, x : [pre \ , \ post[x \backslash E]]; \\ & x := E. \end{aligned}$$

□

B.3.7 Alternation

Law B.22 (alternation "altI") If $pre \Rightarrow (\vee\, i \bullet G_i)$, then

$$\begin{aligned} & w : [pre \ , \ post] \\ \sqsubseteq \quad & \textbf{if } ([\!]\ i \bullet G_i \rightarrow w : [G_i \wedge pre \ , \ post]) \textbf{ fi}. \end{aligned}$$

□

Law B.23 (alternation "altII")

$$
\begin{aligned}
& \{(\bigvee i \bullet G_i)\}\ \mathit{prog} \\
= \;& \textbf{if } ([\!]\ i \bullet G_i \rightarrow \{G_i\}\ \mathit{prog})\ \textbf{fi}.
\end{aligned}
$$

□

Law B.24 (left-distribution of composition over alternation "lda")

$$
\begin{aligned}
& \textbf{if } ([\!]\ i \bullet G_i \rightarrow \mathit{branch}_i)\textbf{fi};\ \ \mathit{prog} \\
= \;& \textbf{if } ([\!]\ i \bullet G_i \rightarrow \mathit{branch}_i;\ \ \mathit{prog})\textbf{fi}.
\end{aligned}
$$

□

Law B.25 (right-distribution of assignment over alternation "lda")

$$
\begin{aligned}
& x := E;\ \ \textbf{if } ([\!]\ i \bullet G_i \rightarrow \mathit{branch}_i)\textbf{fi} \\
= \;& \textbf{if } ([\!]\ i \bullet G_i[x\backslash E] \rightarrow x := E;\ \ \mathit{branch}_i)\textbf{fi}.
\end{aligned}
$$

□

B.3.8 Iteration

Law B.26 (iteration "iter") Let *inv*, the *invariant*, be any predicate; let V, the *variant*, be any integer-valued expression. Then

$$
\begin{aligned}
& w : [\mathit{inv}\ ,\ \mathit{inv} \wedge \neg(\bigvee i \bullet G_i)] \\
\sqsubseteq \;& \textbf{do} \\
& \quad ([\!]\ i \bullet G_i \rightarrow w : [\mathit{inv} \wedge G_i\ ,\ \mathit{inv} \wedge (0 \leq V \leq V_0)]) \\
& \textbf{od}.
\end{aligned}
$$

Neither *inv* nor G_i may contain initial variables. The expression V_0 is $V[w\backslash w_0]$.
□

Law B.27 (iteration single guard "isg") Let *inv*, the *invariant*, be any predicate; let V, the *variant*, be any integer-valued expression. Then

$$
\begin{aligned}
& w : [\mathit{inv}\ ,\ \mathit{inv} \wedge \neg G] \\
\sqsubseteq \;& \textbf{do } G \rightarrow \\
& \quad w : [G\ ,\ \mathit{inv}\ ,\ (0 \leq V \leq V_0)] \\
& \textbf{od}.
\end{aligned}
$$

Neither *inv* nor *G* may contain initial variables.
□

Law B.28 (initialized iteration "ii")

$$\begin{array}{ll} & w : [pre\ ,\ inv \wedge \neg G] \\ \sqsubseteq & w : [pre\ ,\ inv]; \\ & \textbf{do}\ G \rightarrow w : [G \wedge inv\ ,\ inv \wedge (0 \leq V < V_0)]\ \textbf{od}. \end{array}$$

□

B.3.9 Sequential Composition

Law B.29 (sequential composition "scI") For fresh constants X,

$$\begin{array}{ll} & w, x : [pre\ ,\ post] \\ \sqsubseteq & \textbf{con}\ X \bullet \\ & x : [pre\ ,\ mid]; \\ & w, x : [mid[x_0 \backslash X]\ ,\ post[x_0 \backslash X]]. \end{array}$$

The predicate *mid* must not contain initial variables other than x_0.
□

Law B.30 (sequential composition "scII")

$$\begin{array}{ll} & w, x : [pre\ ,\ post] \\ \sqsubseteq & x : [pre\ ,\ mid]; \\ & w, x : [mid\ ,\ post]. \end{array}$$

The predicate *mid* must not contain initial variables; and *post* must not contain x_0.
□

B.3.10 Procedure

Law B.31 (value substitution "vs") If *post* does not contain *f*, then

$$\begin{array}{ll} & w : [pre[f\backslash A]\ ,\ post[f_0\backslash A_0]] \\ \sqsubseteq & [\textbf{value}\ f : T\backslash A] \bullet \\ & w, f : [pre\ ,\ post], \end{array}$$

where A_0 is $A[w\backslash w_0]$.

□

Law B.32 (result substitution "rs") If f does not occur in pre, and neither f nor f_0 occurs in $post$, then

$$\begin{array}{ll} & w, a : [pre\ ,\ post] \\ \sqsubseteq & [\textbf{result}\ f : T\backslash a] \bullet \\ & w, f : [pre\ ,\ post[a\backslash f]]. \end{array}$$

□

Law B.33 (value-result substitution "vrsI") If $post$ does not contain f, then

$$\begin{array}{ll} & w, a : [pre[f\backslash a]\ ,\ post[f_0\backslash a_0]] \\ \sqsubseteq & [\textbf{value result}\ f : T\backslash a] \bullet \\ & w, f : [pre\ ,\ post[a\backslash f]]. \end{array}$$

□

Law B.34 (value-result substitution "vrsII") If $post$ does not contain a, then

$$\begin{array}{ll} & w, a : [pre[f\backslash a]\ ,\ post[f_0, f\backslash a_0, a]] \\ \sqsubseteq & [\textbf{value result}\ f : T\backslash a] \bullet \\ & w, f : [pre\ ,\ post]. \end{array}$$

□

Law B.35 (rename formal parameter "rfp") If l does not occurs in program $prog$, then

$$prog[\textbf{par}\ f : T\backslash A] \quad = \quad prog[f\backslash l][\textbf{par}\ l : T\backslash A].$$

□

Law B.36 (multiple substitution "ms") Provided neither f nor g occurs in F or G,

$$
\begin{aligned}
& prog[\textbf{par1}\ f : T\backslash F][\textbf{par2}\ g : U\backslash G] \\
\sqsubseteq\ & prog[\textbf{par1}\ f : T.\ \textbf{par2}\ g : U\backslash F, G].
\end{aligned}
$$

The substitutions **par1** and **par2** may be any combination of **value**, **result**, and **value result**.

□

B.3.11 Invariant

Law B.37 (remove invariant "ri") Provided w does not occur in inv,

$$w : [pre\ ,\ inv\ ,\ post] \quad \sqsubseteq \quad w : [pre\ ,\ post].$$

□

B.3.12 Skip

Law B.38 (skip command "sk") If $(w = w_0) \wedge pre \Rightarrow post$, then

$$w : [pre\ ,\ post] \quad \sqsubseteq \quad \textbf{skip}.$$

□

Law B.39 (skip composition "skc") For any program $prog$,

$$
\begin{aligned}
& prog;\ \textbf{skip} \\
=\ & \textbf{skip};\ prog \\
=\ & prog.
\end{aligned}
$$

□

Appendix C

A Pascal Program that Computes Even Paragraphs

```
program EvenParagraph(input, output);

const
    maxLength = 46;
    maxWord = 100;

type
    CharArray = packed array [1..maxLength] of char;

    Word =
    record
        word: CharArray;
        length: integer
    end;

    IntegerArray = array [1..maxWord] of integer;

var
    words: array [1..maxWord] of Word;
    total: integer;


(***************************************************)

    procedure ConsumeWhiteSpace;

    var
```

```
   x: char;

   begin
   while not eof and (input^ = ' ') do
       read(x)
   end; { ConsumeWhiteSpace }


(**************************************************)

   procedure ReadWord(var wd: CharArray; var lg: integer);

   var
   x: char;

   begin
   lg := 0;
   while not eof and (input^ <> ' ') do
       if lg < maxLength then begin
           lg := lg + 1;
           read(wd[lg])
       end else
           read(x)
   end; { ReadWord }


(**************************************************)

   procedure ReadInput;

   begin
   ConsumeWhiteSpace;
   total := 0;
   while (total <> maxWord) and not eof do begin
       total := total + 1;
       ReadWord(words[total].word, words[total].length);
       ConsumeWhiteSpace
   end
   end; { ReadInput }


(**************************************************)
```

```
    function max(a, b: integer): integer;

    begin
    if a > b then
        max := a
    else
        max := b
    end; { max }


(****************************************************)

    function min(a, b: integer): integer;

    begin
    if a < b then
        min := a
    else
        min := b
    end; { min }


(****************************************************)

    procedure ComputeMinWasteArray(var mwA: IntegerArray);

    var
    j, n, s, x: integer;

    begin
    j := total;
    mwA[total] := 0;
    while j <> 1 do begin
        j := j - 1;
        n := j + 1;
        s := words[j].length + words[j + 1].length + 1;
        x := max(maxLength - words[j].length, mwA[j + 1]);
        while (n <> total) and (s <= maxLength) do begin
            x := min(x, max(maxLength - s, mwA[n + 1]));
            s := s + words[n + 1].length + 1;
            n := n + 1
        end;
        if s <= maxLength then
            x := 0;
        mwA[j] := x
    end
    end; { ComputeMinWasteArray }


(***************************************************)

    procedure WriteLine(s, f: integer);

    var
    k: integer;

    begin
    write(words[s].word: words[s].length);
    k := s + 1;
    while k <= f do begin
        write(' ');
        write(words[k].word: words[k].length);
        k := k + 1
    end;
    writeln
    end; { WriteLine }


(***************************************************)

    procedure WriteEven(mwA: IntegerArray);

    var
    i, j, s: integer;

    begin
    i := 1;
    while i <> total + 1 do begin
        j := i;
        s := maxLength - words[j].length;
        while (j <> total) and
            ((s > mwA[1]) or (mwA[j + 1] > mwA[1])) do begin
            j := j + 1;
            s := s - words[j].length - 1
        end;
        WriteLine(i, j);
        i := j + 1
    end
    end; { WriteEven }


(****************************************************)

    procedure WriteParagraph;

    var
    minWasteArray: IntegerArray;

    begin
    if total >= 1 then begin
        ComputeMinWasteArray(minWasteArray);
        WriteEven(minWasteArray)
    end
    end; { WriteParagraph }


(****************************************************)

begin
    total := 0;
    ReadInput;
    WriteParagraph
end. { EvenParagraph }
```